Secure, Confidential Blockchains Providing High Throughput and Low Latency

Thèse N° 7101

Présentée le 27 septembre 2019

à la Faculté informatique et communications
Unité du Professeur Ford
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Eleftherios KOKORIS KOGIAS

Acceptée sur proposition du jury

Prof. C. González Troncoso, présidente du jury
Prof. B. A. Ford, directeur de thèse
Prof. C. Cachin, rapporteur
Prof. G. Danezis, rapporteur
Prof. J.-P. Hubaux, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Any fool can criticize, complain, and condemn –and most fools do.
But it takes character and self-control to be understanding and forgiving.
– Dale Carnegie

To those who believed in me. . . , especially my family.

Acknowledgments

First, I would like to thank my advisor, Prof. Bryan Ford. Bryan allowed me to work independently without ever feeling pressure to deliver results, nevertheless being willing to give me feedback when I asked for it. He taught me a lot including what good research is, how it should be done and especially how to present it in a convincing way. Without his support and advice, this journey would not have been possible.

I also want to thank Prof. Christian Cachin, Prof. George Danezis and Prof. Jean-Pierre Hubaux for their feedback on this dissertation, and Prof. Carmela Troncoso for presiding over my thesis committee.

Many thanks for all our discussions to all the members of the Decentralized and Distributed Systems Laboratory: Jeff Allen, Ludovic Barman, Cristina Basescu, Gaylor Bosson, Simone Colombo, Kelong Cong, Georgia Fragkouli, David Froelicher, Noemien Kocher, and Stevens Le Blond. A big thanks to Linus Gasser and Nicolas Gailly for helping with the translation of my abstract too. During my Ph.D., I was also lucky enough to meet some special people at DEDIS, whom I consider good friends. Cey, Kirill, Philipp, and Ewa made the days and nights of the Ph.D. much more enjoyable and for that, I am deeply grateful. I would also like to thank them for being great collaborators, and for contributing to the research in this thesis. Finally, a big thanks to Holly Cogliati-Bauereis, Angela Devenonge, Maggy Escandari, Anouk Hein, and Patricia Hjelt, for making my life significantly easier by helping with everything related to (among other things) administrative tasks and English editing.

Many people outside the lab were also very important in these four years. A special thanks to George, with whom we've been good friends and collaborators since our undergraduate school, to my "cousin" Marios, and to my neighbor Lana. You were the people that would put up with my complaining and be available for a quick beer or "one last slope" run. I would also like to thank Irene, Dusan, Hermina, Helena and the rest that are already mentioned that started the Ph.D. trip with me and surprisingly managed to stay close and travel together despite our crazy schedules. Also, a big thanks to my friends in Zurich: Maria, Dimitris, Marios, Zeta, Chrysa, Patrica, and Stelios who made it a second home during my stay and visits there.

I would also like to thank all of my friends in Greece and abroad, who make me feel like there is always a friend around the corner, especially when I travel to conferences. There are too

many of them to thank and too much to write for each one, but they all know who they are and how much they have done for me, being there for me for many years now, and enduring my unpredictable schedule and rare visits back home.

Last, but in no way least, I would like to thank my family, my parents Yianni and Katerina, my sister Marita, and my brother George, for shaping the person I am today and for supporting me unconditionally in my every step. They have always made the time I spend in Greece relaxing and memorable and although I feel like I have a lot of homes, the home of my heart is wherever they are.

My Ph.D. work has been supported by the EDIC Ph.D. Fellowship and the IBM Ph.D. Fellowship.

Lausanne, April 10th 2019 Eleftherios Kokoris-Kogias

Abstract

One of the core promises of blockchain technology is that of enabling trustworthy data dissemination in a trustless environment. What current blockchain systems deliver, however, is *slow* dissemination of *public* data, rendering blockchain technology unusable in settings where latency, transaction capacity or data confidentiality is important. In this thesis, we focus on providing solutions on two of the most pressing problems blockchain technology currently faces: *scalability* and *data confidentiality*.

To address the scalability issue, we present OMNILEDGER, a novel scale-out distributed ledger that preserves long-term security under permissionless operation. It ensures security and correctness by using a bias-resistant public-randomness protocol for choosing large, statistically representative shards that process transactions, and by introducing an efficient cross-shard commit protocol that atomically handles transactions affecting multiple shards.

To enable the secure sharing of confidential data we present CALYPSO, the first fully decentralized auditable access-control framework for secure blockchain-based data sharing which builds upon two abstractions. First, *on-chain secrets* enable collective management of (verifiably shared) secrets under a Byzantine adversary where an access-control blockchain enforces user-specific access rules and a secret-management cothority administers encrypted data. Second, *skipchain-based identity and access management* enables efficient administration of dynamic, sovereign identities and access policies and, in particular, permits clients to maintain long-term relationships with respect to the evolving user identities thanks to the trust-delegating forward links of skipchains.

In order to build OMNILEDGER and CALYPSO, we first build a set of tools for efficient decentralization, which are presented in Part II of this dissertation. These tools can be used in decentralized and distributed systems to achieve (1) scalable consensus (BYZCOIN), (2) bias-resistant distributed randomness creations (RANDHOUND), and (3) relationship-keeping between independently updating communication endpoints (SKIPPER). Although we use these tools in the scope of this thesis, they can be (and already have been) used in a far wider scope.

Keywords: Blockchain, Scalability, Privacy, Scale-Out, Secure permissionless Distributed Ledger, Consensus, Atomic Commit, Confidentiality, Decentralization.

Résumé

L'une des principales promesses de la technologie blockchain est de permettre la diffusion de données de confiance dans un environnement sans confiance. Cependant, les systèmes de blockchain actuels fournissent une diffusion *lente* des données *publiques*, rendant la technologie inutilisable dans les environnements où la latence, la capacité de transaction, où la confidentialité des données sont d'une importance primordiale. Dans cette thèse, nous nous concentrons sur deux problèmes que la technologie blockchain rencontre actuellement : *le déploiement à grande échelle* et *la confidentialité des données*.

Pour permettre le déploiement à grande échelle de la blockchain, nous presentons une nouvelle, scale-out, blockchain, qui guarantie la sécurité à long terme dans un environnement libre et ouvert. OMNILEDGER assure la sécurité et l'integrité des données en utilisant deux mechanismes complementaires. Le premier est un protocole generant des nombres aléatoires et qui est resistant a la manipulation en utilisant de grandes partition de l'ensemble des participants représentatifs sur le plan statistique. Le deuxieme est un protocole efficace de validation des données entre partitions qui gère atomiquement les transactions affectant plusieurs partitions.

Pour permettre le partage sécurisé de données confidentielles, nous présentons CALYPSO, le premier système de contrôle d'accès entièrement décentralisé et vérifiable pour le partage sécurisé de données basé sur une blockchain. CALYPSO repose sur deux abstractions : Tout d'abord, *on-chain secrets* permet la gestion collective de secrets (partagés de manière vérifiable), dans un environnement byzantin. Dans cet environnement, nous utilisons une blockchain de gestion d'accès qui applique des règles d'accès spécifiques aux utilisateurs qui souhaitent acceder aux données stockées, et une autorité de gestion de secret qui administre des données cryptées. Deuxièmement, *skipchain-based identity and access management* permet d'administrer efficacement un systeme d'identités dynamiques et souveraines et les politiques d'accès aux données. *skipchain-based identity and access management* permet notamment aux clients de maintenir des relations à long terme en ce qui concerne l'identité des utilisateurs en constante évolution, grâce à la délégation de confiance en utilisant les liens en avance des skipchains. *skipchain-based identity and access management* permet notamment d'imposer des droits d'accès et de délégations sur des identités dont l'évolution est gérée grâce aux liens cryptographiques présents dans la skipchain.

Afin de construire OMNILEDGER et CALYPSO, nous avons d'abord créé un ensemble d'outils présentés dans la deuxième partie de cette thèse permettant de developper des systemes decentralisés éfficaces à grande échelle. Nous utilisons ensuite ces outils pour réaliser (1)

un protocole de consensus à grande échelle (ByzCoin), (2) une protocole de generation de nombres aléatoires publiquement verifiable, et résistant aux manipulations (RandHound), et (3) un protocole maintien des relations entre les systèmes de mise à jour de manière indépendante (Skipper). Bien que nous utilisions ces outils dans le cadre de la blockchain pour cette thèse, ils peuvent être (et ont déjà été) utilisés dans d'autres cadres.
Mots-clés : Blockchain, Confidentialité, Registre Distribué en mode "sans permission", Consensus, Atomic Commit, Décentralisation.

Contents

List of Figures

List of Tables

Part I Introduction and Background

1 Introduction

1.1 Motivation

Blockchain technology has emerged a decade ago with Bitcoin [191], an open, self-regulating cryptocurrency build on top of the idea of decentralization. A blockchain can be defined as an append-only distributed log that stores an ordered set of log-entries, also known as transactions. Transactions are grouped into batches or *blocks* that form a cryptographic hash-chain, hence the name blockchain. These transactions typically encode valid updates to the *state* of the blockchain. Given that (i) a blockchain is an ordered set of transactions, (ii) these transactions cause the state to change, and (iii) the state is stored among a distributes set of servers, we can say that blockchains are concrete implementations of the state-machine replication abstraction [220].

The most widely adopted blockchain application is cryptocurrencies that provide an alternative to fiat currencies and traditional banking and promise to become the infrastructure for a new generation of Internet interactions, including anonymous online payments [218], remittance [239], and transacting of digital assets [11]. Currently, there are thousands of cryptocurrencies with a market capitalization exceeding $260B. Blockchain technology, however, has outgrown the initial application of cryptocurrencies and is currently promising to disrupt whole industries such as finance [251], insurance [239], healthcare [170], logistics [239], and governments [203]. Claims that are backed by more than $15B of funding.

This "hype" and abundance of funding, has lead to a frenzy of "white" papers (*i.e.,* non-peer-reviewed design documents for blockchain start-ups) claiming to have a solution for almost any problem humanity has, but usually ignoring most prior work. Although blockchain technology made decentralization mainstream, the idea of decentralization existed decades before. The decentralization concept was already part of how the Internet was supposed to function and it was concretely introduced by David Chaum in 1981 [57] in order to propose an alternative to centralized systems that inherently have single points of failure or compromise. In this thesis we also go back in time, before the inception of Bitcoin and study decentralized systems such as Practical Byzantine Fault Tolerance (PBFT) [55], threshold

cryptosystems [110, 135, 222, 227, 229], and multi-signatures [221, 39] in order to use them as fundamental building blocks to providing solutions for two of the most pressing problems of the blockchain ecosystem [42]: scalability and data privacy.

1.1.1 The Scalability Challenge

Currently, all open blockchain consensus protocols deployed have two challenging limitations as far as scalability is concerned: confirmation latency and transaction throughput. First, the core consensus used in open blockchains requires synchronous communication [199] to guarantee persistence of the blocks. As a result, clients may need to wait up to 60 minutes [191], before accepting a transaction as valid, in order to be confident that they have the same view as the rest of the network.

Second, every fully participating node in the network must process every transaction in order for the system to be really decentralized. There should be no central party that guarantees the security (and more specifically the consistency) of the blockchain, and instead every single participating node is responsible for securing the system. In principle, this means that every single participating node needs to see, process, and store a copy of every transaction and of the entire state. As a result, the throughput of Bitcoin is limited to 7 transactions per second, 3-4 orders of magnitude lower than traditional payment networks such as VISA.

1.1.2 The Data Privacy Challenge

The second major challenge this thesis studies stems from non cryptocurrency-based blockchain applications in sectors such as finance [251], (personalized) healthcare [170], insurance [239] or e-democracy [203]. These applications regularly tend to assume shared access to sensitive data between independent, mutually distrustful parties. However, current blockchains fail to manage confidential data securely and efficiently on-chain.

Most blockchain systems ignore this problem [253, 139] whereas others implement naive solutions such as encrypting the information with the public keys of the intended recipients and publishing the ciphertexts on Bitcoin [109]. The problem here is that once these ciphertexts are public the application no longer has control over who can access them and when, because the access control is enforced statically during encryption. To prevent these issues, many systems [138, 256, 15] fall back to semi-centralized solutions that record access requests, permissions, and hashes of the data on-chain, but manage the secret data off-chain keeping the encryptions and the decryption keys in a centralized or distributed storage. This approach makes the storage provider a single point of failure, as it can deny access, even for legitimate requests, or decrypt the confidential data using the untrusted keys undetected.

1.2 Contributions and Publications

This thesis addresses the two challenges above by first building the necessary tools for efficient decentralization and then proposing scalable and private blockchains both in the permissionless setting, where the set of participants is unknown and in the permissioned setting, where the set of participants is well-defined.

More specifically, the contributions of this thesis are the following:

- We introduce BYZCOIN, the first open (permissionless) blockchain system that provides non-probabilistic consensus by forming rolling Proof-of-Work committees that maintain a Bitcoin-like blockchain. As part of BYZCOIN, we design a strongly-consistent consensus algorithms that can be run efficiently by hundreds of servers and provides high throughput and low confirmation latency.
- We show how to generate bias-resistant distributed randomness in a large scale by leveraging our scalable consensus and verifiable secret sharing. The core idea is to collect randomness from many highly-available, but not always honest clusters of nodes and combine it to one randomness output. Given that at least one cluster is collectively honest, we can guarantee that the combined randomness is unpredictable and unbiasable.
- We introduce skipchains, a doubly-linked blockchain that can be used to track long-term relationships in decentralized and distributed systems. The core idea behind skipchains is that of *forward links*. Forward links are generated by an authoritative committee that manages some service by signing the membership of succeeding committees. This forward link effectively delegates trust from the authoritative committee to its successor and enables clients to securely and efficiently obtain authoritative data without always needing to be online.
- We combine the three tools of decentralization mentioned above (*i.e.,* consensus, randomness, and relationship-tracking) to introduce OMNILEDGER, the first fully decentralized and long-term secure sharded blockchain. The core idea behind OMNILEDGER is to let participants validate only a subset of the transactions, while still having all transactions validated by a big enough set of validators to prevent inconsistencies. As a result, validation can be done in parallel and the system *scales out*, meaning that the more popular the system becomes the higher the throughput it can provide.
- We design CALYPSO, an extension that enables confidential data sharing over programmable blockchains. More specifically, we enable a writer to shares confidential data with a set of readers (*e.g.,* a company) and guarantee that if any reader accesses the data, the writer will hold proof-of-access (*i.e.,* forensics) and that if a reader has the right to access the data, then the data will be delivered upon request. Furthermore, we enable the dynamic evolution of the readers' identities and of the data access-policies with the deployment of skipchains for sovereign, decentralized identity management.

- Finally, we study the same problems of horizontal scalability (scale-out) and confidential data sharing in the permissioned setting where weaker trust assumptions enable us to design simpler protocols. We map this new design on top of HyperLedger Fabric [10], the most widely used permissioned blockchain until the moment of writing.

Most of the work presented in this thesis is based on the following co-authored publications:

- 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 the 25th USENIX Security Symposium, 2016
- Eleftherios Kokoris-Kogias, Linus Gasser, Ismail Khoffi, Philipp Jovanovic, Nicolas Gailly, and Bryan Ford. "Managing Identities Using Blockchains and CoSi".
  In Hot Topics in Privacy Enhancing Technologies (HotPETs), 2016
- 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 the 38th IEEE Symposium on Security and Privacy, 2017
- Bryan Ford, Linus Gasser, Eleftherios Kokoris Kogias, Philipp Jovanovic "Cryptographically Verifiable Data Structure Having Multi-Hop Forward and Backwards Links and Associated Systems and Methods".
  US Patent App. 15/618,653, 2018[1]
- Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta and Bryan Ford. "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding".
  In the 39th IEEE Symposium on Security and Privacy, 2018
- Eleftherios Kokoris-Kogias, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Linus Gasser, Philipp Jovanovic, Ewa Syta, and Bryan Ford. "CALYPSO: Auditable Sharing of Private Data over Blockchains."
  Under Submission
- Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. "Channels: Horizontal Scaling and Confidentiality on Permissioned Blockchains".
  In the 23rd European Symposium on Research in Computer Security, 2018

1.3 Organization and Structure

This thesis is organized into four parts and ten chapters.

[1] The paper "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds" [195] used skipchains as part of general software update framework and is not part of this thesis.

Chapter 2 introduces the systems we use as building blocks for this thesis. First, we cover committee-based (or quorum-based) agreement protocols necessary for building our scalable consensus algorithm. Then we introduce Bitcoin and one of its improvements, Bitcoin-NG, that define the problem and the environment we set this thesis. Finally, we introduce threshold cryptosystems, that help remove single points of failure or compromise from the systems we build and enable full decentralization.

Chapter 3 commences the second part, where we introduce our constructions that enable efficient and secure decentralization. More specifically, in this chapter, we built a scalable consensus in an open and adversarial environment and show that it provides a faster and more secure alternative to Bitcoin, albeit under a different but realistic adversarial model.

Chapter 4 describes our second tool; randomness. More specifically, we show how hard it can be to build a fully-decentralized randomness beacon whose randomness is unpredictable and unbiasable. Then we provide two different implementations of such a randomness beacon who scale to hundreds of participating servers.

Chapter 5 provides the last tool necessary to build our fully decentralized solutions, Skipchains. With skipchains, we enable clients to maintain long-term relationships with dynamically changing sets of servers (who provide some service) without the need to trust a third-party for indirection.

Part 3 of this dissertation describes three different systems that solve the two challenges described earlier.

First, in Chapter 6, we see OMNILEDGER. OMNILEDGER is the first horizontally scalable, secure and fully decentralized blockchain system. We achieve this by leveraging the tools of Part 2 and combining them with a 20-year old idea from database systems: sharding.

Then, in Chapter 7, we switch gears and focus on the data privacy challenge. We tackle this challenge by using threshold secret sharing in order to create a highly available vault for storing encryption keys, which can be coupled with blockchain systems in order to provide fully decentralized access-control on confidential data.

Finally, in Chapter 8, we look into the above problem from the angle of permissioned blockchains and formalize an asset management application, which concurrently scales horizontally and provides confidentiality among the different state-spaces of the full system.

The last part of this thesis surveys the related work in Chapter 9 and provides concluding remarks and directions for future work in Chapter 10.

2 Background

2.1 Committee-based Agreement Protocols

2.1.1 Scalable Agreement via Collective Signing

CoSi [243] is a protocol for scalable collective signing, which enables an authority or *leader* to request that statements be publicly validated and *(co-)signed* by a decentralized group of *witnesses*. Each protocol run yields a *collective signature* having size and verification cost comparable to an individual signature, but which compactly attests that both the leader and its (perhaps many) witnesses observed and agreed to sign the statement.

To achieve scalability, CoSi combines Schnorr multi-signatures [221] with communication trees that are long used in multicast protocols [54]. Initially, the protocol assumes that signature verifiers know the public keys of the leader and those of its witnesses, all of which combine to form a well-known aggregate public key. For each message to be collectively signed, the leader then initiates a CoSi four-phase protocol round that requires two round-trips over the communication tree between the leader and its witnesses:

1. **Announcement:** The leader broadcasts an announcement of a new round down the communication tree. The announcement can optionally include the message M to be signed, otherwise M is sent in phase three.
2. **Commitment:** Each node picks a random secret and uses it to compute a Schnorr commitment. In a bottom-up process, each node obtains an aggregate Schnorr commitment from its immediate children, combines those with its own commitment, and passes a further-aggregated commitment up the tree.
3. **Challenge:** The leader computes a collective Schnorr challenge using a cryptographic hash function and broadcasts it down the communication tree, along with the message M to sign, if the latter has not already been sent in phase one.

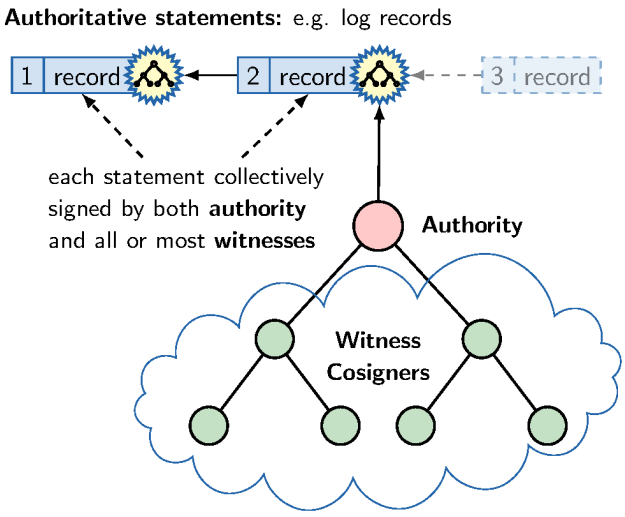

Figure 2.1 – CoSi protocol architecture

4. **Response:** Using the collective challenge, all nodes compute an aggregate response in a bottom-up fashion that mirrors the commitment phase.

The result of this four-phase protocol is the production of a standard Schnorr signature that requires about 64 bytes, using the Ed25519 elliptic curve [27], and that anyone can verify against the aggregate public key nearly as efficiently as the verification of an individual signature. Practical caveats apply if some witnesses are offline during the collective signing process: in this case, the CoSi protocol may proceed, but the resulting signature grows to include metadata verifiably documenting which witnesses did and did not co-sign. We refer to the CoSi paper for details [243]. Drijvers et al. [85] has shown a subtle attack on the Schnorr version of CoSi and for this reason we suggest the use of BLS [39] signatures instead.

2.1.2 Practical Byzantine Fault Tolerance

The *Byzantine Generals' Problem* [162, 200] refers to the situation where the malfunctioning of one or several components of a distributed system prevents the latter from reaching an agreement. Pease et al. [200] show that $3f+1$ participants are necessary to be able to tolerate f faults and still reach consensus. The *Practical Byzantine Fault Tolerance (PBFT)* algorithm [55] was the first efficient solution to the Byzantine Generals' Problem that works in weakly synchronous environments such as the Internet. PBFT offers both *safety* and *liveness* provided that the above bound applies, *i.e.*, that at most f faults among $3f+1$ participants, hold.

PBFT triggered a surge of research on Byzantine state-machine replication algorithms with various optimizations and trade-offs [1, 154]. Every round of PBFT has three distinct phases as illustrated in Figure 2.2. In the first, *pre-prepare* phase, the current primary node or *leader* announces the next record that the system should agree upon. On receiving this pre-prepare,

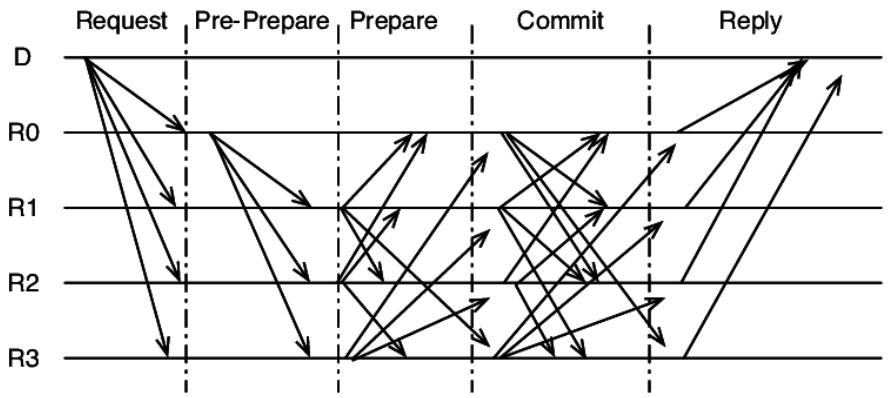

Figure 2.2 – PBFT Communication Pattern. D is the client and R_i are replicas

every node validates the correctness of the proposal and multicasts a *prepare* message to the group. The nodes wait until they collect a quorum of $(2f+1)$ prepare messages and publish this observation with a *commit* message. Finally, they wait for a quorum of $(2f+1)$ commit messages to make sure that enough nodes have recorded the decision.

PBFT relies upon a correct leader to begin each round and proceeds if a two-thirds quorum exists; consequently, the leader is an attack target. For this reason, PBFT has a view-change protocol that ensures liveness in the face of a faulty leader. All nodes monitor the leader's actions and if they detect either malicious behavior or a lack of progress, initiate a view-change. Each node independently announces its desire to change leaders and stops validating the leader's actions. If a quorum of $(2f+1)$ nodes decides that the leader is faulty, then the next leader in a well-known schedule takes over.

PBFT has its limitations, which make it unsuitable for permissionless blockchains. First, it assumes a fixed, well-defined group of replicas, thus contradicting Bitcoin's basic principle of being decentralized and open for anyone to participate. Second, each PBFT replica normally communicates directly with every other replica during each consensus round, resulting in $O(n^2)$ communication complexity: This is acceptable when n is typically 4 or not much more, but becomes impractical if n represents hundreds or thousands of Bitcoin nodes. Third, after submitting a transaction to a PBFT service, a client must communicate with a super-majority of the replicas in order to confirm the transaction has been committed and to learn its outcome, making secure transaction *verification* unscalable.

2.2 Blockchain Systems

2.2.1 Bitcoin

At the core of Bitcoin [191] rests the so-called *blockchain*, a public, append-only database maintained by *miners* and serving as a global ledger of all transactions ever issued. Transac-

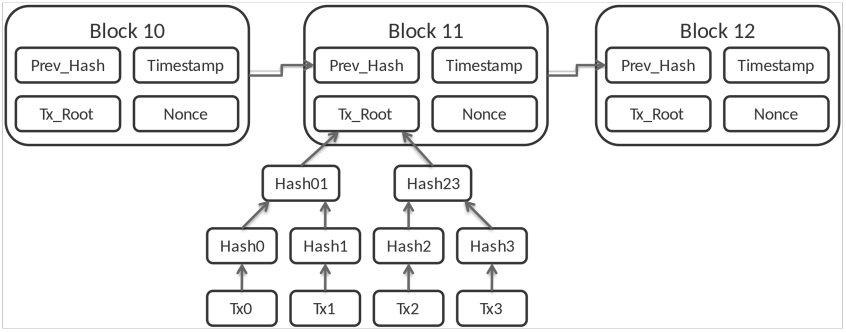

Figure 2.3 – Bitcoin's Blockchain network data

tions are bundled into *blocks* and validated by a *proof-of-work*. A valid Proof-of-Work block is one whose cryptographic hash has d leading zero bits, where the difficulty parameter d is adjusted periodically such that new blocks are mined about every ten minutes on average.

Each block in a blockchain includes a Merkle tree [179] of new transactions to be committed, and a cryptographic hash chaining to the last valid block, thereby forming the blockchain. Upon successfully forming a new block with a valid proof-of-work, a miner broadcasts the new block to the rest of the miners, who (when behaving properly) accept the new block if it extends a valid chain strictly longer than any other chain they have already seen. An illustration of Bitcoin's blockchain data model can be seen in Figure 2.3.

Bitcoin's decentralized consensus and security derive from an assumption that a majority of the miners, measured in terms of *hash power* or ability to solve hash-based proof-of-work puzzles, follows these rules and always attempts to extend the longest existing chain. As soon as a set of miners controlling the majority of the network's hash power approves a given block by mining on top of it, the block remains embedded in any future chain [191]. Bitcoin's security is guaranteed by the fact that this majority will be extending the legitimate chain faster than any corrupt minority that might try to rewrite history or double-spend currency. However, Bitcoin's consistency guarantee is only probabilistic and synchrony dependent, which leads to two fundamental problems.

First, multiple miners might find distinct blocks with the same parent before the network has reached consensus. Such a conflict is called a *fork*, an inconsistency that is temporarily allowed until one of the chains is extended yet again. Subsequently, all well-behaved miners on the shorter chain(s) switch to the new longest one. All transactions appearing only in the rejected block(s) are invalid and must be resubmitted for inclusion into the winning blockchain. This means that Bitcoin clients who want high certainty that a transaction persists (*e.g.*, that they

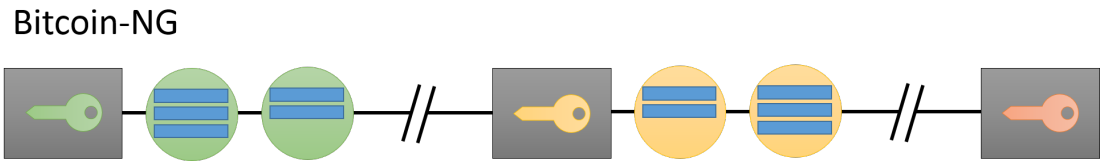

Figure 2.4 – Bitcoin-NG's Blockchain. Slow keyblocks include the keys of the miners who produce fast microblocks that include transactions

have irrevocably received a payment) must wait not only for the next block but for several blocks thereafter, thus increasing the transaction latency. As a rule of thumb [191], a block is considered as permanently added to the blockchain after about 6 new blocks have been mined on top of it, for a confirmation latency of 60 minutes on average.

Second, the Bitcoin block size is currently limited to 1 MB. This limitation in turn results in an upper bound on the number of transactions per second (TPS) the Bitcoin network can handle, estimated to be an average of 7 TPS. For comparison, Paypal handles 500 TPS and VISA even 4000 TPS. An obvious solution to enlarge Bitcoin's throughput is to increase the size of its blocks. Unfortunately, this solution also increases the probability of forks due to higher propagation delays and the risk of double-spending attacks [112]. Bitcoin's liveness and security properties depend on forks being relatively rare. Otherwise, the miners would spend much of their effort trying to resolve multiple forks [113], or in the extreme case, completely centralize Bitcoin [95]

2.2.2 Bitcoin-NG

Bitcoin-NG [95] makes the important observation that Bitcoin blocks serve two different purposes: (1) election of a leader who decides how to resolve potential inconsistencies, and (2) verification of transactions. Due to this observation, Bitcoin-NG proposes two different block types (see Figure 2.4): *Keyblocks* are generated through mining with proof-of-work and are used to securely elect leaders, at a moderate frequency, such as every 10 minutes as in Bitcoin. *Microblocks* contain transactions, require no proof-of-work, and are generated and signed by the elected leader. This separation enables Bitcoin-NG to process many microblocks between the mining of two keyblocks, enabling transaction throughput to increase.

Bitcoin-NG, however, retains many drawbacks of Bitcoin's consistency model. Temporary forks due to near-simultaneous keyblock mining, or deliberately introduced by selfish or malicious miners, can still throw the system into an inconsistent state for 10 minutes or more. Further, within any 10-minute window, the current leader could still intentionally fork or rewrite history and invalidate transactions. If a client does not wait several tens of minutes (as in Bitcoin) for transaction confirmation, he is vulnerable to double-spend attacks by the current leader or by another miner who forks the blockchain. Although Bitcoin-NG includes disincentives for such behavior, these disincentives amount at most to the "mining value" of

the keyblock (coinbase rewards and transaction fees): Thus, leaders are both able and have incentives to double-spend on higher-value transactions.

Consequently, although Bitcoin-NG permits higher transaction throughput, it does not solve Bitcoin's consistency weaknesses. Nevertheless, Bitcoin-NG's decoupling of keyblocks from microblocks is an important idea that we build on in Chapter 3 to support high-throughput and low-latency transactions in ByzCoin.

2.3 Threshold Cryptosystems

2.3.1 Secret Sharing

The notion of secret sharing was introduced independently by Blakely [33] and Shamir [227] in 1979. An (t, n)-secret sharing scheme, with $1 \leq t \leq n$, enables a dealer to share a secret a among n trustees such that any subset of t honest trustees can reconstruct a whereas smaller subsets cannot. This means in other words that the sharing scheme can withstand up to $t-1$ malicious participants.

In the case of Shamir's scheme, the dealer evaluates a degree $t-1$ polynomial s at positions $i > 0$ and each share $s(i)$ is handed out to a trustee. The important observation here is that only if a threshold of t honest trustees works together then the shared secret $a = s(0)$ can be recovered (through polynomial interpolation).

2.3.2 Verifiable Secret Sharing

The downside of these simple secret sharing schemes is that they assume an honest dealer which might not be realistic in some scenarios. Verifiable secret sharing (VSS) [59, 99] adds verifiability to those simple schemes and thus enables trustees to verify if shares distributed by a dealer are consistent, that is, if any subset of a certain threshold of shares reconstructs the same secret. VSS has a wide range of applications such as distributed key generation, threshold signatures, and threshold encryption schemes. We describe these schemes below and use them at multiple protocols of this thesis. For all the systems below we assume the existence of a cyclic group $\mathcal{G}$ of prime order p and of a generator G of $\mathcal{G}$.

2.3.3 Threshold Signing

TSS [235] is a distributed (t, n)-threshold Schnorr signature scheme. TSS allows any subset of t signers to produce a valid signature. During setup, all n trustees use VSS to create a long-term shared secret key x and a public key $X = G^x$. To sign a statement S, the n trustees first use VSS to create a short-term shared secret v and a commitment $V = G^v$ and then compute the challenge $c = \mathsf{H}(V \parallel S)$. Afterwards, each trustee i uses his shares v_i and x_i of v and x, respectively, to create a partial response $r_i = v_i - cx_i$. Finally, when t out of n trustees

collaborate they can reconstruct the response r through Lagrange interpolation. The tuple (c, r) forms a regular Schnorr signature on S, which can be verified against the public key X.

2.3.4 Distributed Key Generation

A Distributed Key Generation (DKG) protocol removes the dependency on a trusted dealer from the secret sharing scheme by having every trustee run a secret sharing round. In essence, a (n, t) DKG [141] protocol allows a set of n servers to produce a secret whose shares are spread over the nodes such that any subset of servers of size greater than t can reveal or use the shared secret, while smaller subsets do not have any knowledge about the secret. Pedersen has offered the first DKG scheme [201] based upon the regular discrete logarithm problem without any trusted party.

We provide here a summery of how Pedersen DKG works. We assume the existence of n servers participating in the DKG and each of them is in possession of a private-public key pair. The list of public keys is publicly known. The Pedersen DKG assume the existence of a private channel between any pairs of participants and a broadcast channel available to all participants. Furthermore, this DKG scheme works in a synchronous setting where an upper bound on the communication delay is fixed and known in advance. While this may be restrictive in today's global Internet era, a sufficient large timeout can simulate such synchrony. If such a synchrony assumption is too strong, Aniket et al. [141] have provided a partially synchronous DKG variant, that can substitute the one described below.

Key Generation The protocol follows several steps in order to securely generate a distributed key:

1. Each party P_i chooses a random polynomial $f_i(z)$ over $\mathbb{Z}_p^*$ of degree t:

$$f_i(z) = a_{i0} + a_{i1} * z + \cdots + a_{it} * z^t$$

2. Each P_i computes the individual secret shares $s_{ij} = f_i(j) \mod p$ for all $j \in \{1, \ldots, n\}$ and sends s_{ij} to party P_j using a confidential point-to-point channel. We denote a_{i0} by x_i, the individual secret contribution to the distributed private key.

3. Each P_i broadcasts the commitment to the coefficients $A_{ik} = G^{a_{ik}}$ for all $k \in \{0, \ldots, k\}$ to all other participants. We denote A_{i0} by X_i, the individual public contribution to the distributed public key.

4. Each P_j verifies the share received from the other parties by checking, for all $i \in \{1, \ldots, n\}$:

$$G^{s_{ij}} = \prod_{k=0}^{t} (A_{ik})^{j^k} \mod p \tag{2.1}$$

If the check fails for an index i, P_j broadcasts a *complaint* against P_i.

5. For each complaining party P_j, party P_i reveals the corresponding share s_{ij} matching (2.1). If any of the revealed shares fails this equation, P_i is disqualified. We define the set QUAL to be the set of non-disqualified parties.

6. The public key is defined as $X = \prod_{i \in QUAL} X_i$. Each party P_j sets his share of the secret to $x_j = \sum_{i \in QUAL} s_{ij} \mod p$. The public verification values are computed as $A_k = \prod_{i \in QUAL} A_{ik}$. The distributed secret key is defined as $x = \sum_{i \in QUAL}^{t} x_j * \lambda_i$ where λ_i is the i-th Lagrange coefficient.

2.3.5 Threshold ElGamal Encryption

The ElGamal cryptosystem [89] is an asymmetric cryptographic system enabling one to encrypt a message without any interactions needed from the destination. In this thesis, we focus on the CCA-2 (Adaptive chosen-ciphertext adversary) secure decentralized variant introduced by Shoup et al [229], Specifically, we focus on the TDH2 scheme from [229] as a *chosen ciphertext secure* threshold encryption system, which relies on the *decisional Diffie Hellman* problem (DDH). We briefly recap the TDH2 system here.

To be at all useful, the key-holding (decryption) servers should not decrypt everything that comes its way and give it to just anybody, but should implement some kind of useful decryption policy. To implement such a policy securely, in addition to chosen-ciphertext security, one needs an additional property: the ability to attach a label to the ciphertext during the encryption process. Such a label L is a bit-string that contains information that can be used by the decryption servers to determine if the decryption request is authorized. One can think of the label as being a part of the ciphertext so that changing the label changes the ciphertext; security against chosen ciphertext attack would then imply, in particular, that one cannot subvert the third party's policy by simply swapping labels.

For simplicity of describing the scheme, we assume the message and labels are l-bit strings. TDH2 requires the existence of three distinct hash functions:

$$H_1 : \mathcal{G} \rightarrow \{0,1\}^l,$$

$$H_2 : \{0,1\}^l \times \{0,1\}^l \times \mathcal{G}^4 \rightarrow \mathbb{Z}_p,$$

$$H_3 : \mathcal{G}^3 \rightarrow \mathbb{Z}_p$$

1. *Key Generation:* The set of participating servers collectively run a DKG protocol and

each server ends up with having one share x_i of the distributed secret. The participating server also decides on a random additional generator $\bar{G}$ ($\bar{G}$ is the same for all servers).

2. *Encryption:* The algorithm to encrypt a message $m \in \{0,1\}^l$ and a label L runs as follow. The encryptor chooses $r, s \in \mathbb{Z}_p$ at random.

$$c = H_1(X^r) \oplus m, u = G^r, w = G^s, \bar{u} = \bar{G}^r, \bar{w} = \bar{G}^s,$$
$$e = H_2(c, L, u, w, \bar{u}, \bar{w}), f = s + re$$

The ciphertext is $\psi = (c, L, u, \bar{u}, e, f)$.

3. *Label Extraction:* Given an appropriately encoded ciphertext $(c, L, u, \bar{u}, e, f)$, the label extraction algorithm simply outputs L.

4. *Decryption:* Decryption server i checks the following equation given the ciphertext $\psi = (c, L, u, \bar{u}, e, f)$:

$$e \stackrel{?}{=} H_2(c, L, u, w, \bar{u}, \bar{w}) \tag{2.2}$$
$$with\ w = G^f / u^e, and\ \bar{w} = \bar{G}^f / \bar{u}^e$$

If the check fails, the server i refuses to decrypt the ciphertext. Otherwise, the server i chooses a $s_i \in \mathbb{Z}_p$ at random and computes the following:

$$u_i = u^{x_i}, \hat{u}_i = u^{s_i}, h_i = G^{s_i},$$
$$e_i = H_3(u_i, \hat{u}_i, \hat{h}_i), f_i = s_i + x_i * e_i$$

The server i outputs its decryption share (i, u_i, e_i, f_i).

5. *Share Verification:* This step verifies if a decryption share from a server i is valid or not given the distributed public key and the ciphertext ψ. First, the ciphertext must pass the equation 2.2. If it does not, then the share i is invalid. Otherwise, the share is considered valid if and only if it is of the form (i, u_i, e_i, f_i) and

$$e_i \stackrel{?}{=} H_3(u_i, \hat{u}_i, \hat{h}_i),$$
$$where\ \hat{u}_i = u^{f_i} / u_i^{e_i}, \hat{h}_i = G^{f_i} / h^{e_i}$$

6. *Combining shares:* The share combining algorithm takes as input the verification key VK, a ciphertext c, and a full set of valid decryption shares of c. If the test 2.2 does not hold, then the algorithm outputs "?" (all the decryption shares are considered invalid). So we can assume that the set of decryption shares is of the form:

$$(i, u_i, e_i, f_i) : i \in S$$

where S has cardinality k. Then the message m can be recovered using Lagrange interpolation

$$m = H_1(\prod_{i \in S} u_i^{\delta_{0i}}) \oplus c$$

with δ being the Lagrange coefficients.

2.3.6 Publicly Verifiable Secret Sharing

PVSS [222] is a (t, n)-secret sharing scheme that enables third-party verifiability of secret shares without revealing any information on the shares or the secret. To obtain this property, PVSS uses non-interactive zero-knowledge (NIZK) proofs for equality of discrete logarithms [56, 100, 106].

Let $H \neq G$ be another generator of $\mathcal{G}$. Unless stated otherwise, we assume that $i \in \{1, \ldots, n\}$ and $j \in \{0, \ldots, t-1\}$. PVSS runs in three steps:

1. **Share Distribution (Dealer).** To distribute a secret among the trustees, the dealer executes the following steps:

   (a) Select coefficients $a_j \in_R \mathbb{Z}_q^*$ of a $t-1$ degree secret sharing polynomial

   $$s(x) = \sum_{j=0}^{t-1} a_j x^j \, .$$

   The secret to-be-shared is $S_0 = G^{s(0)} = G^{a_0}$.

   (b) Compute share commitments $H^{s(i)}$, encrypted shares $\widehat{S}_i = X_i^{s(i)} = G^{s(i)x_i}$, and polynomial commitments $A_j = H^{a_j}$. Note that $H^{s(i)}$ can be recovered from A_j as follows:

   $$\prod_{j=0}^{t-1} A_j^{(i^j)} = H^{\sum_{j=0}^{t-1} a_j i^j} = H^{s(i)} \, .$$

   (c) Create encryption consistency proofs $\widehat{P}_i$ which enable to publicly verify that $\log_H H^{s(i)} = \log_{X_i} \widehat{S}_i$. Therefore, pick $v_i \in_R \mathbb{Z}_q$, compute $V_{i1} = H^{v_i}$, $V_{i2} = X_i^{v_i}$, $c = \mathsf{H}((H^{s(i)})_{i \in M}, (\widehat{S}_i)_{i \in M}, (V_{i1}, V_{i2})_{i \in M})$, and $r_i = v_i - s(i)c \mod q$, and set $\widehat{P}_i = (c, r_i, V_{i1}, V_{i2})$.

   (d) Publish $\widehat{S}_i$, $\widehat{P}_i$, and A_j.

2. **Share Decryption (Trustee).** To decrypt his share, trustee i executes the following steps:

(a) Verify $\widehat{S}_i$ against $\widehat{P}_i$ by reconstructing $H^{s(i)}$ from A_j and by checking that

$$V_{i1} \stackrel{?}{=} H^{r_i}(H^{s(i)})^c = H^{v_i - s(i)c} H^{s(i)c} = H^{v_i}$$
$$V_{i2} \stackrel{?}{=} X_i^{r_i}(\widehat{S}_i)^c = X_i^{v_i - s(i)c} X_i^{s(i)c} = X_i^{v_i} .$$

(b) If the previous step fails, abort. Otherwise, decrypt $\widehat{S}_i$ using x_i and obtain $S_i = (\widehat{S}_i)^{x_i^{-1}}$.

(c) Create a decryption consistency proof P_i which enables to publicly verify that $\log_G X_i = \log_{S_i} \widehat{S}_i$. Therefore, pick $v_i' \in_R \mathbb{Z}_q$, compute $V_{i1}' = G^{v_i'}$, $V_{i2}' = (S_i)^{v_i'}$, $c_i' = \mathsf{H}(X_i, \widehat{S}_i, V_{i1}', V_{i2}')$, $r_i' = v_i' - x_i c_i'$, and set $P_i = (c_i', r_i', V_{i1}', V_{i2}')$.

(d) Publish S_i and P_i.

3. **Secret Recovery (Dealer).** To reconstruct the secret S_0, the dealer executes the following steps:

(a) Verify S_i against P_i by checking that

$$V_{i1}' \stackrel{?}{=} G^{r_i'} X_i^{c_i'} = G^{v_i' - x_i c_i'} G^{x_i c_i'} = G^{v_i'}$$
$$V_{i2}' \stackrel{?}{=} (S_i)^{r_i'} (\widehat{S}_i)^{c_i'} = (S_i)^{v_i' - x_i c_i'} (S_i)^{x_i c_i'} = (S_i)^{u_i'}$$

and discard S_i if the verification fails.

(b) Suppose w.l.o.g, for $1 \le i \le t$, that shares S_i are valid. Reconstruct secret S_0 by Lagrange interpolation

$$\prod_{i=1}^{t} (S_i)^{\lambda_i} = \prod_{i=1}^{t} (G^{s(i)})^{\lambda_i} = G^{\sum_{i=1}^{t} s(i)\lambda_i} = G^{s(0)} = S_0$$

where $\lambda_i = \prod_{j \neq i} \frac{j}{j-i}$ is a Lagrange coefficient.

Part II Tools for Efficient Decentralization

3 Scalable, Strongly-Consistent Consensus for Bitcoin

3.1 Introduction

The original Bitcoin paper argues that transaction processing is secure and irreversible, as long as the largest colluding group of miners represents less than 50% of total computing capacity and at least about one hour has elapsed. This high transaction-confirmation latency limits Bitcoin's suitability for real-time transactions. Later work revealed additional vulnerabilities to transaction reversibility, double-spending, and strategic mining attacks [96, 113, 128, 140, 194, 13].

The key problem is that Bitcoin's consensus algorithm provides only probabilistic consistency guarantees. Strong consistency could offer cryptocurrencies three important benefits. First, all miners instantly agree on the validity of blocks, without wasting computational power resolving inconsistencies (*forks*). Second, clients need not wait for extended periods to be certain that a submitted transaction is committed; as soon as it appears in the blockchain, the transaction can be considered confirmed. Third, strong consistency provides *forward security*: as soon as a block has been appended to the blockchain, it stays there forever. Although increasing the consistency of cryptocurrencies has been suggested before [70, 77, 176, 224, 246], existing proposals give up Bitcoin's decentralization, and/or introduce new and non-intuitive security assumptions, and/or lack experimental evidence of performance and scalability.

This chapter introduces ByzCoin, a Bitcoin-like cryptocurrency enhanced with strong consistency, based on the principles of the well-studied Practical Byzantine Fault Tolerance (PBFT) [55] algorithm. ByzCoin addresses four key challenges in bringing PBFT's strong consistency to cryptocurrencies: (1) open membership, (2) scalability to hundreds of replicas, (3) proof-of-work block conflicts, and (4) transaction commitment rate.

PBFT was not designed for scalability to large consensus groups: deployments and experiments often employ the minimum of four replicas [154], and generally, have not explored scalability levels beyond 7 [55] or 16 replicas [69, 122, 1]. ByzCoin builds PBFT atop CoSi [243],

a collective signing protocol that efficiently aggregates hundreds or thousands of signatures. Collective signing reduces both the costs of PBFT rounds and the costs for "light" clients to verify transaction commitment. Although CoSi is not a consensus protocol, BYZCOIN implements Byzantine consensus using CoSi signing rounds to make PBFT's *prepare* and *commit* phases scalable.

PBFT normally assumes a well-defined, closed group of replicas, conflicting with Bitcoin's open membership and use of proof-of-work to resist Sybil attacks [84]. BYZCOIN addresses this conflict by forming consensus groups dynamically from *windows* of recently mined blocks, giving recent miners *shares* or voting power proportional to their recent commitment of hash power. Lastly, to reduce transaction processing latency we adopt the idea from Bitcoin-NG [95] to decouple transaction verification from block mining.

Experiments with a prototype implementation of BYZCOIN show that a consensus group formed from approximately the past 24 hours of successful miners (144 miners) can reach consensus in less than 20 seconds, on blocks of Bitcoin's current maximum size (1MB). A larger consensus group formed from one week of successful miners (1008) reached consensus on an 8MB block in 90 seconds, showing that the system scales both with the number of participants and with the block size. For the 144-participant consensus group, with a block size of 32MB, the system handles 974 transactions per second (TPS) with a 68-second confirmation latency. These experiments suggest that BYZCOIN can handle loads higher than PayPal and comparable with Visa.

BYZCOIN is still a proof-of-concept with several limitations. First, BYZCOIN does not improve on Bitcoin's proof-of-work mechanism; finding a suitable replacement [14, 105, 147, 254, 43] is an important but orthogonal area for future work. Like many BFT protocols in practice [62, 122], BYZCOIN is vulnerable to slowdown or temporary DoS attacks that Byzantine nodes can trigger. Although a malicious leader cannot violate or permanently block consensus, he might temporarily exclude minority sets ($< \frac{1}{3}$) of victims from the consensus process, depriving them of rewards, and/or attempt to censor transactions. BYZCOIN guarantees security only against attackers who consistently control less than a third (not 50%) of consensus group shares – though Bitcoin has analogous weaknesses accounting for. Finally, BYZCOIN's security is at present analyzed only informally (Section 3.4).

BYZCOIN makes the following key contributions:

- We use Collective Signing [243] to scale BFT protocols to large consensus groups and enable clients to verify operation commitments efficiently.
- We present the first demonstrably practical Byzantine consensus protocol supporting not only static consensus groups but also dynamic membership proportional to proof-of-work as in Bitcoin.
- We demonstrate experimentally that a strongly-consistent cryptocurrency can increase

Bitcoin's throughput by two orders of magnitude, with a transaction confirmation-latency under one minute.

- We find through security analysis (Section 3.4) that BYZCOIN can mitigate several known attacks on Bitcoin provided no attacker controls more than $\frac{1}{4}$ of hash power.

The remainder of the chapter is organized as follows. Section 3.2 details the BYZCOIN protocol. Section 3.3 describes an evaluation of our prototype implementation of BYZCOIN. Section 3.4 informally analyzes BYZCOIN's security and Section 3.6 concludes.

3.2 ByzCoin Design

This section presents BYZCOIN with a step-by-step approach, starting from a simple "strawman" combination of PBFT and Bitcoin. From this strawman, we progressively address the challenges of determining consensus group membership, adapting Bitcoin incentives and mining rewards, making the PBFT protocol scale to large groups and handling block conflicts and selfish mining.

3.2.1 System Model

BYZCOIN is designed for untrustworthy networks that can arbitrarily delay, drop, re-order or duplicate messages. To avoid the Fischer-Lynch-Paterson impossibility [102], we assume the network has a weak synchronyproperty [55]. The BYZCOIN system is comprised of a set of N block miners that can generate key-pairs, but there is no trusted public-key infrastructure. Each node i has a limited amount of *hash power* that corresponds to the maximum number of block-header hashes the node can perform per second.

At any time t a subset of miners controlled by a malicious attacker that are considered faulty. Byzantine miners can behave arbitrarily, diverting from the protocol and colluding to attack the system. The remaining honest miners follow the prescribed protocol. We assume that the total hash power of all Byzantine nodes is less than $\frac{1}{4}$ of the system's total hash power at any time, since proof-of-work-based cryptocurrencies become vulnerable to selfish mining attacks by stronger adversaries [96].

3.2.2 Strawman Design: PBFTCoin

For simplicity, we begin with PBFTCoin, an unrealistically simple protocol that naively combines PBFT with Bitcoin, then gradually refine it into BYZCOIN.

For now, we simply assume that a group of $n = 3f + 1$ PBFT replicas, which we call *trustees*, has been fixed and globally agreed upon upfront, and that at most f of these trustees are faulty. As in PBFT, at any given time, one of these trustees is the *leader*, who proposes transactions and

drives the consensus process. These trustees collectively maintain a Bitcoin-like blockchain, collecting transactions from clients and appending them via new blocks, while guaranteeing that only one blockchain history ever exists and that it can never be rolled back or rewritten. Prior work has suggested essentially such a design [70, 77], though without addressing the scalability challenges it creates.

Under these simplifying assumptions, PBFTCoin guarantees safety and liveness, as at most f nodes are faulty and thus the usual BFT security bounds apply. However, the assumption of a fixed group of trustees is unrealistic for Bitcoin-like decentralized cryptocurrencies that permit open membership. Moreover, as PBFT trustees authenticate each other via non-transferable symmetric-key MACs, each trustee must communicate directly with most other trustees in every round, thus yielding $O(n^2)$ communication complexity.

Subsequent sections address these restrictions, transforming PBFTCoin into ByzCoin in four main steps:

1. We use Bitcoin's proof-of-work mechanism to determine consensus groups dynamically while preserving Bitcoin's defense against Sybil attacks.
2. We replace MAC-authenticated direct communication with digital signatures to make authentication transferable and thereby enabling sparser communication patterns that can reduce the normal case communication latency from $O(n^2)$ to $O(n)$.
3. We employ scalable collective signing to reduce per-round communication complexity further to $O(\log n)$ and reduce typical signature verification complexity from $O(n)$ to $O(1)$.
4. We decouple transaction verification from leader election to achieve a higher transaction throughput.

3.2.3 Opening the Consensus Group

Removing PBFTCoin's assumption of a closed consensus group of trustees presents two conflicting challenges. On the one hand, conventional BFT schemes rely on a well-defined consensus group to guarantee safety and liveness. On the other hand, Sybil attacks [84] can trivially break any open-membership protocol involving security thresholds, such as PBFT's assumption that at most f out of $3f+1$ members are honest.

Bitcoin and many of its variations employ a mechanism already suited to this problem: proof-of-work mining. Only miners who have dedicated resources are allowed to become a member of the consensus group. In refining PBFTCoin, we adapt Bitcoin's proof-of-work mining into a *proof-of-membership* mechanism. This mechanism maintains the "balance of power" within the BFT consensus group over a given fixed-size sliding *share window*. Each time a miner finds a new block, it receives a *consensus group share*, which proves the miner's membership

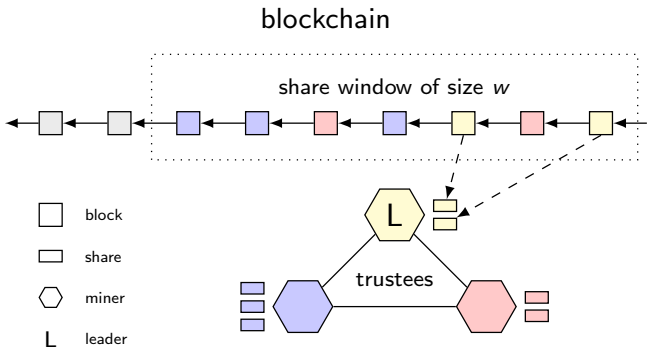

Figure 3.1 – Valid shares for mined blocks in the blockchain are credited to miners

in the group of trustees and moves the share window one step forward. Old shares beyond the current window expire and become useless for purposes of consensus group membership. Miners holding no more valid shares in the current window lose their membership in the consensus group, hence they are no longer allowed to participate in the decision-making.

At a given moment in time, each miner wields "voting power" of a number of shares equal to the number of blocks the miner has successfully mined within the current window. Assuming collective hash power is relatively stable, this implies that within a window, each active miner wields a number of shares statistically proportionate to the amount of hash power that the miner has contributed during this time period.

The size w of the share window is defined by the average block-mining rate over a given time frame and influences certain properties such as the resilience of the protocol to faults. For example, if we assume an average block-mining rate of 10 minutes and set the duration of the time frame to one day (or one week), then $w = 144$ ($w = 1008$). This mechanism limits the membership of miners to recently active ones, which prevents the system from becoming unavailable due to too many trustees becoming inactive over time, or from miners aggregating many shares over an extended period and threatening the balance in the consensus group. The relationship between blocks, miners, and shares is illustrated in Figure 3.1.

Mining Rewards and Transaction Fees As we can no longer assume voluntary participation as in PBFTCoin's closed group of trustees, we need an incentive for nodes to obtain shares in the consensus group and to remain active. For this purpose, we adopt Bitcoin's existing incentives of mining rewards and transaction fees. But instead of these rewards all going to the miner of the most recent block we split a new block's rewards and fees across all members of the current consensus group, in proportion to the number of shares each miner holds. As a consequence, the more hash power a miner has devoted within the current window, hence the more shares the miner holds, the more revenue the miner receives during payouts in the current window. This division of rewards also creates incentives for consensus group members to remain live and participate, because they receive their share of the rewards for new blocks

only if they continually participate, in particular, contributing to the prepare and commit phases of each BFT consensus round.

3.2.4 Replacing MACs by Digital Signatures

In our next refinement step towards ByzCoin, we tackle the scalability challenge resulting from PBFT's typical communication complexity of $O(n^2)$, where n is the group size. PBFT's choice of MAC-authenticated all-to-all communication was motivated by the desire to avoid public-key operations on the critical transaction path. However, the cost of public-key operations has decreased due to well-optimized asymmetric cryptosystems [27], making those costs less of an issue.

By adopting digital signatures for authentication, we are able to use sparser and more scalable communication topologies, thus enabling the current leader to collect and distribute third-party verifiable evidence that certain steps in PBFT have succeeded. This removes the necessity for all trustees to communicate directly with each other. With this measure we can either enable the leader to collect and distribute the digital signatures or let nodes communicate in a chain [122], reducing the normal-case number of messages from $O(n^2)$ to $O(n)$.

3.2.5 Scalable Collective Signing

Even with signatures providing transferable authentication, the need for the leader to collect and distribute – and for all nodes to verify – many individual signatures per round can still present a scalability bottleneck. Distributing and verifying tens or even a hundred individual signatures per round might be practical. If we, however, want consensus groups with a thousand or more nodes (*e.g.,* representing all blocks successfully mined in the past week), it is costly for the leader to distribute 1000 digital signatures and wait for everyone to verify them. To tackle this challenge, we build on the CoSi protocol [243] for collective signing. CoSi does not directly implement consensus or BFT, but it offers a primitive that the leader in a BFT protocol can use to collect and aggregate prepare and commit messages during PBFT rounds.

We implement a single ByzCoin round by using two sequential CoSi rounds initiated by the current leader (*i.e.,* the owner of the current view). The leader's announcement of the first CoSi round (phase 1 in Section 2.1.1) implements the *pre-prepare* phase in the standard PBFT protocol (Section 2.1.2). The collective signature resulting from this first CoSi round implements the PBFT protocol's *prepare* phase, in which the leader obtains attestations from a two-thirds super-majority quorum of consensus group members that the leader's proposal is safe and consistent with all previously-committed history.

As in PBFT, this prepare phase ensures that a proposal *can be* committed consistently, but by itself, it is insufficient to ensure that the proposal *will be* committed. The leader and/or some number of other members could fail before a super-majority of nodes learn about the successful prepare phase. The ByzCoin leader, therefore, initiates a second CoSi round to

implement the PBFT protocol's *commit* phase, in which the leader obtains attestations from a two-thirds super-majority that all the signing members witnessed the successful result of the prepare phase and made a positive commitment to remember the decision. This collective signature, resulting from this second CoSi round, effectively attests that a two-thirds super-majority of members not only considers the leader's proposal "safe" but promises to remember it, indicating that the leader's proposal is fully committed.

In cryptocurrency terms, the collective signature resulting from the prepare phase provides a proof-of-acceptance of a proposed block of transactions. This block is not yet committed, however, since a Byzantine leader that does not publish the accepted block could double-spend by proposing a conflicting block in the next round. In the second CoSi commit round, the leader announces the proof-of-acceptance to all members, who then validate it and collectively sign the block's hash to produce a collective commit signature on the block. This way, a Byzantine leader cannot rewrite history or double-spend, because by counting arguments at least one honest node would have to sign the commit phase of both histories, which an honest node by definition would not do.

The use of CoSi does not affect the fundamental principles or semantics of PBFT but improves its scalability and efficiency in two main ways. First, during the commit round where each consensus group member must verify that a super-majority of members have signed the prior prepare phase, each participant generally needs to receive only an $O(1)$-size rather than $O(n)$-size message, and to expend only $O(1)$ rather than $O(n)$ computation effort by verifying a single collective signature instead of n individual ones. This benefit directly increases the scalability and reduces the bandwidth and computation costs of consensus rounds themselves.

A second benefit is that after the final CoSi commit round has completed, the final resulting collective commit signature serves as a typically $O(1)$-size proof, which anyone can verify in $O(1)$ computation time that a given block – hence any transaction within that block – has been irreversibly committed. This secondary scalability-benefit might in practice be more important than the first, because it enables "light clients" who neither mine blocks nor store the entire blockchain history to verify quickly and efficiently that a transaction has committed, without requiring active communication with or having to trust any particular full node.

3.2.6 Decoupling Transaction Verification from Leader Election

Although ByzCoin so far provides a scalable guarantee of strong consistency, thus ensuring that clients need to wait only for the next block rather than the next several blocks to verify that a transaction has committed, the time they still have to wait *between* blocks can, nevertheless, be significant: up to 10 minutes using Bitcoin's difficulty tuning scheme. Whereas ByzCoin's strong consistency might in principle make it "safe" from a consistency perspective to increase block mining rate, doing so could still exacerbate liveness and other performance issues, as in Bitcoin [191]. To enable lower client-perceived transaction latency, therefore, we build on the idea of Bitcoin-NG [95] to decouple the functions of transaction verification from block

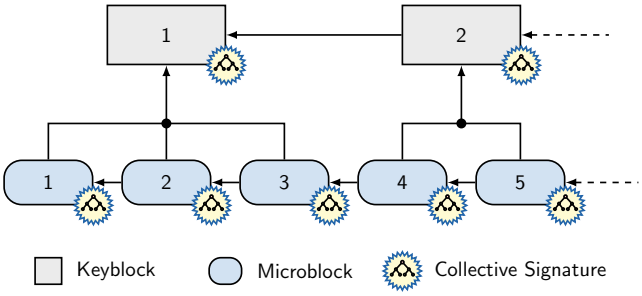

Figure 3.2 – ByzCoin blockchain: Two parallel chains store information about the leaders (keyblocks) and the transactions (microblocks)

mining for leader election and consensus group membership.

As in Bitcoin-NG, we use two different kinds of blocks. The first, *microblocks* or transaction blocks, represent transactions to be stored and committed. The current leader creates a new microblock every few seconds, depending on the size of the block, and uses the CoSi-based PBFT protocol above to commit and collectively sign it. The other type of block, *keyblocks*, are mined via proof-of-work as in Bitcoin and serve to elect leaders and create shares in ByzCoin's group membership protocol as discussed earlier in Section 3.2.3. As in Bitcoin-NG, this decoupling allows the current leader to propose and commit many microblocks that contain many smaller batches of transactions, within one ≈ 10-minute keyblock mining period. Unlike Bitcoin-NG, in which a malicious leader could rewrite history or double-spend within this period until the next keyblock, ByzCoin ensures that each microblock is irreversibly committed regardless of the current leader's behavior.

In Bitcoin-NG one blockchain includes both types of blocks, which introduces a race condition for miners. As microblocks are created, the miners have to change the header of their keyblocks to mine on top of the latest microblock. In ByzCoin, in contrast, the blockchain becomes two separate parallel blockchains, as shown in Figure 3.2. The main blockchain is the keyblock chain, consisting of all mined blocks. The microblock chain is a secondary blockchain that depends on the primary to identify the era in which every microblock belongs to, *i.e.*, which miners are authoritative to sign it and who is the leader of the era.

Microblocks A microblock is a simple block that the current consensus group produces every few seconds to represent newly-committed transactions. Each microblock includes a set of transactions and a collective signature. Each microblock also includes hashes referring to the previous microblock and keyblock: the former to ensure total ordering, and the latter indicating which consensus group window and leader created the microblock's signature. The microblock's hash is collectively signed by the corresponding consensus group.

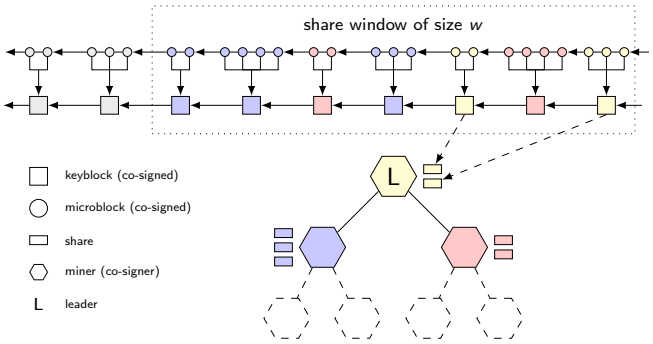

Figure 3.3 – Overview of the full ByzCoin design

Keyblocks Each keyblock contains a proof-of-work, which is used to determine consensus group membership via the sliding-window mechanism discussed earlier, and to pay signers their rewards. Each newly-mined keyblock defines a new consensus group, and hence a new set of public keys with which the next era's microblocks will be collectively signed. Since each successive consensus group differs from the last in at most one member, PBFT ensures the microblock chain's consistency and continuity across this group membership change provided at most f out of $3f+2$ members are faulty.

Bitcoin-NG relies on incentives to discourage the next leader from accidentally or maliciously "forgetting" a prior leader's microblocks. In contrast, the honest super-majority in a ByzCoin consensus group will refuse to allow a malicious or amnesiac leader to extend any but the most recently-committed microblock, regardless of which (current or previous) consensus group committed it. Thus, although competing keyblock conflicts may still appear, these "forks" cannot yield an inconsistent microblock chain. Instead, a keyblock conflict can at worst temporarily interrupt the PBFT protocol's liveness, until it is resolved as mentioned in Section 3.2.6.

Decoupling transaction verification from leader election and consensus group evolution in this way brings the overall ByzCoin architecture to completion, as illustrated in Figure 3.3. Subsequent sections discuss further implications and challenges this architecture presents.

Keyblock Conflicts and Selfish Mining

PBFT's strong consistency by definition does not permit inconsistencies such as forks in the microblock chain. The way the miners collectively decide how to resolve keyblock conflicts, however, can still allow selfish mining [96] to occur as in Bitcoin. Worse, if the miners decide randomly to follow one of the two blocks, then keyblock forks might frequently lead to PBFT liveness interruptions as discussed above, by splitting miners "too evenly" between competing keyblocks. Another approach to deciding between competing keyblocks is to impose a deterministic priority function on their hash values, such as "smallest hash wins." Unfortunately, this practice can encourage selfish mining.

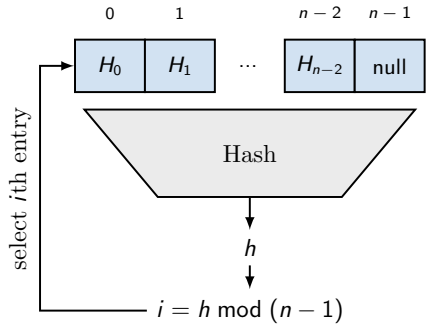

Figure 3.4 – Deterministic fork resolution in BYZCOIN

One way to break a tie without helping selfish miners is to increase the entropy of the output of the deterministic prioritization function. We implement this idea using the following algorithm. When a miner detects a keyblock fork, it places all competing blocks' header hashes into a sorted array, from low to high hash values. The array itself is then hashed, and the final bit(s) of this hash determine which keyblock wins the fork.

This solution, shown in Figure 3.4, also uses the idea of a deterministic function applied to the blocks, thus requiring no voting. Its advantage is that the input of the hash function is partially unknown before the fork occurs, thus the entropy of the output is high and difficult for an attacker to be able to optimize. Given that the search space for possible conflict is as big as the search space for a new block, trying to decide if a block has better than 50% probability of winning the fork is as hard as finding a new block.

Leader Election and PBFT View Changes

Decoupling transaction verification from the block-mining process comes at a cost. So far we have assumed, as in PBFT, that the leader remains fixed unless he crashes or misbehaves. If we keep this assumption, then this leader gains the power of deciding which transactions are verified, hence we forfeit the fairness-enforcing benefit of Bitcoin's leader election. To resolve this issue, every time a keyblock is signed, BYZCOIN forces a mandatory PBFT view-change to the keyblock's miner. This way the power of verifying transactions in blocks is assigned to the rightful miner, who has an era of microblock creation from the moment his keyblock is signed until the next keyblock is signed.

When a keyblock conflict occurs, more than one such "mandatory" view-change occurs, with the successful miners trying to convince other participants to adopt their keyblock and its associated consensus group. For example, in a keyblock fork, one of the two competing keyblocks wins the resolution algorithm described above. However, if the miner of the "losing" block races to broadcast its keyblock and more than 33% honest miners have already committed to it before learning about the competing keyblock, then the "winning" miner is too late and the system either commits to the first block or (in the worst case) loses liveness temporarily as

discussed above. This occurs because already-committed miners will not accept a mandatory view-change except to a keyblock that represents their committed state and whose microblock chain extends all previously-committed microblocks. Further analysis of how linearizability is preserved across view-changes may be found in the original PBFT paper [55].

Tree Creation in BYZCOIN

Once a miner successfully creates a new keyblock, he needs to form a CoSi communication tree for collective signing, with himself as the leader. If all miners individually acknowledge this keyblock to transition to the next view, this coordination normally requires $O(N)$ messages. To avoid this overhead at the beginning of each keyblock round, the miners autonomously create the next round's tree bottom-up during the previous keyblock round. This can be done in $O(1)$ by using the blockchain as an array that represents a full tree.

This tree-building process has three useful side-effects. First, the previous leader is the first to get the new block, hence he stops creating microblocks and wasting resources by trying to sign them. Second, in the case of a keyblock conflict, potential leaders use the same tree, and the propagation pattern is the same; this means that all nodes will learn and decide on the conflict quickly. Finally, in the case of a view change, the new view will be the last view that worked correctly. So if the leader of the keyblock i fails, the next leader will again be the miner of keyblock $i-1$.

3.2.7 Tolerating Churn and Byzantine Faults

In this section, we discuss the challenges of fault tolerance in BYZCOIN, particularly tree failures and maximum tolerance for Byzantine faults.

Tree Fault Tolerance

In CoSi, there are multiple different mechanisms that allow substantial fault-tolerance. Furthermore, the strict membership requirements and the frequent tree changes of BYZCOIN increase the difficulty for a malicious attacker with less than around 25% of the total hash power to compromise the system. A security analysis, however, must assume that a Byzantine adversary is able to get the correct nodes of the BYZCOIN signing tree so that it can compromise the liveness of the system by a simple DoS.

To mitigate this risk, we focus on recent Byzantine fault tolerance results [122], modifying BYZCOIN so that the tree communication pattern is a normal-case performance optimization that can withstand most malicious attacks. But when the liveness of the tree-based BYZCOIN is compromised, the leader can return to non-tree-based communication until the end of his era.

The leader detects that the tree has failed with the following mechanism: After sending the block to his children, the leader starts a timer that expires before the view-change timer. Then he broadcasts the hash of the block he proposed and waits. When the nodes receive this message they check if they have seen the block and either send an ACK or wait until they see the block and then send the ACK. The leader collects and counts the ACKs, to detect if his block is rejected simply because it never reaches the witnesses. If the timer expires or a block rejection arrives before he receives two-thirds of the ACKs, the leader knows that the tree has failed and reverts to a flat BYZCOIN structure before the witnesses assume that he is faulty.

As we show in Section 3.3, the flat BYZCOIN structure can still quickly sign keyblocks for the day-long window (144 witnesses) while maintaining a throughput higher than Bitcoin currently supports. Flat BYZCOIN is more robust to faults, but increases the communication latency back to $O(n)$. Furthermore, if all faults ($\lfloor \frac{N}{3} \rfloor$) are consecutive leaders, this can lead back to a worst case $O(n^2)$ communication latency.

Membership Churn and BFT

After a new leader is elected, the system needs to ensure that the first microblock of the new leader points to the last microblock of the previous leader. Having $2f+1$ supporting votes is not enough. This occurs because there is the possibility that an honest node lost its membership when the new era started. Now in the worst case, the system has f Byzantine nodes, f honest nodes that are up to date, f slow nodes that have a stale view of the blockchain, and the new leader that might also have a stale view. This can lead to the leader proposing a new microblock, ignoring some signed microblocks and getting $2f+1$ support (stale+Byzantine+his own). For this reason, the first microblock of an era needs $2f+2$ supporting signatures. If the leader is unable to obtain them, this means that he needs to synchronize with the system, *i.e.,* he needs to find the latest signed microblock from the previous roster. He asks all the nodes in his roster, plus the node that lost its membership, to sign a latest-checkpoint message containing the hash of the last microblock. At this point in time, the system has $3f+2$ ($3f+1$ of the previous roster plus the leader) members and needs $2f+1$ honest nodes to verify the checkpoint, plus an honest leader to accept it (a Byzantine leader will be the $f+1$ fault and compromise liveness). Thus, BYZCOIN can tolerate f fails in a total of $3f+2$ nodes.

3.3 Performance Evaluation

In this section, we discuss the evaluation of the BYZCOIN prototype and our experimental setup. The main question we want to evaluate is whether BYZCOIN is usable in practice without incurring large overheads. In particular, we focus on consensus latency and transaction throughput for different parameter combinations.

3.3.1 Prototype Implementation

We implemented ByzCoin in Go and made it publicly available on GitHub.[1] ByzCoin's consensus mechanism is based on the CoSi protocol with Ed25519 signatures [27] and implements both flat- and tree-based collective signing layouts as described in Section 3.2. For comparison, we also implemented a conventional PBFT consensus algorithm with the same communication patterns as above and a consensus algorithm that uses individual signatures and tree-based communication.

To simulate consensus groups of up to 1008 nodes, we oversubscribed the available 36 physical machines (see below) and ran up to 28 separate ByzCoin processes on each server. Realistic wide-area network conditions are mimicked by imposing a round-trip latency of 200 ms between any two machines and a link bandwidth of 35 Mbps per simulated host. Note that this simulates only the connections between miners of the consensus group and not the full Bitcoin network. Full nodes and clients are not part of the consensus process and can retrieve signed blocks only after consensus is reached. Since Bitcoin currently is rather centralized and has only a few dozen mining pools [13], we assume that if/when decentralization happens, all miners will be able to support these rather constrained network requirements.

The experimental data to form microblocks was taken by ByzCoin clients from the actual Bitcoin blockchain. Both micro- and keyblocks are fully transmitted and collectively signed through the tree and are returned to the clients upon request together with the proof. Verification of block headers is implemented but transaction verification is only emulated to avoid further measurement distortion through oversubscribed servers. A similar practice is used in Shadow Bitcoin [184]. We base our emulation both on measurements [113] of the average block-verification delays (around 200 ms for 500 MB blocks) and on the claims of Bitcoin developers [31] that as far as hardware is concerned Bitcoin can easily verify 4000 TPS. We simulate a linear increase of this delay proportional to the number of transactions included in the block. Because of the communication pattern of ByzCoin, the transaction-verification cost delays only the leaf nodes. By the time the leaf nodes finish the block verification and send their vote back to their parents, all other tree nodes should have already finished the verification and can immediately proceed. For this reason, the primary delay factor is the transmission of the blocks that needs to be done $\log N$ sequential times.

We ran all our experiments on DeterLab [80] using 36 physical machines, each having four Intel E5-2420 v2 CPUs and 24 GB RAM and being arranged in a star-shaped virtual topology.

CoSi Pipelining Collective signing [243] is done in four different phases per round, namely announce, response, challenge, and commit. In ByzCoin the announce and commit phases of CoSi can be performed in advance before the block to be committed is available, since the proposed block can be sent to the signers in the challenge phase. This enables us to

[1] https://github.com/dedis/cothority

		t_i	t_{i+1}	t_{i+2}	t_{i+3}	t_{i+4}
B_k	prepare	An/Co	Ch/Re			
	commit		An/Co	Ch/Re		
B_{k+1}	prepare		An/Co	Ch/Re		
	commit			An/Co	Ch/Re	
B_{k+2}	prepare			An/Co	Ch/Re	
	commit				An/Co	Ch/Re

Table 3.1 – BYZCOIN pipelining for maximum transaction-throughput; B_k denotes the microblock signed in round k, An/Co the Announce-Commit and Ch/Re the Challenge-Response round-trips of CoSi

pipeline two rounds so that the announce/commit phases of BYZCOIN's commit round are piggybacked on the challenge and response messages of the prepare round. This pipelining reduces latency by one round-trip over the communication tree. Looking into the normal execution of BYZCOIN, this pipeline can be extended so that an optimal throughput of one signed microblock per round-trip is produced. A sample execution can be seen in Table 3.1.

3.3.2 Evaluation

The main question we want to evaluate is whether BYZCOIN is usable in practice without incurring large overhead. We evaluated keyblock and microblock signing for an increasing number of consensus group members. Further, we evaluated BYZCOIN's latency for an increasing microblock size and an increasing number of consensus group members, for all implemented consensus algorithms. We then compare Bitcoin with the flat and tree-based versions of BYZCOIN to investigate the maximum throughput that each variant can achieve. In another experiment, we investigate the latency of signing single transactions and larger blocks. Finally, we demonstrate the practicality of BYZCOIN, as far as latency for securing transactions is concerned, from a client's point of view.

3.3.3 Consensus Latency

The first two experiments focus on how microblock commitment latency scales with the consensus group size and with the number of transactions per block.

Consensus Group Size Comparison

This experiment focuses on the scalability of BYZCOIN's BFT protocol in terms of the consensus group size. The number of unique miners participating in a consensus group is limited by the number of membership shares in the window (Section 3.2.3), but can be smaller if some miners hold multiple shares (*i.e.*, successfully mined several blocks) within the same window.

We ran experiments for Bitcoin's maximum block size (1 MB) with a variable number of

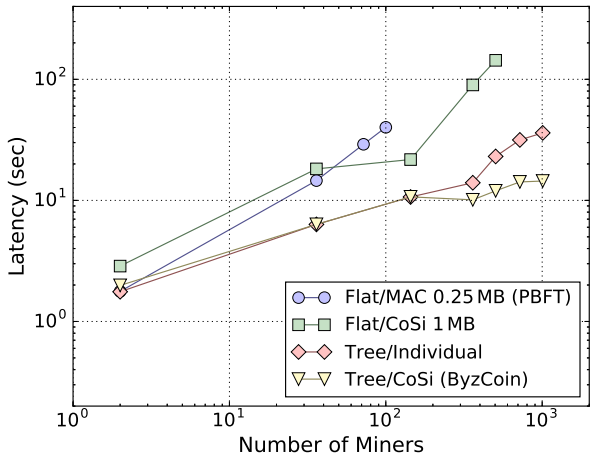

Figure 3.5 – Influence of the consensus group size on the consensus latency

participating hosts. Every time we increased the number of hosts, we also increased the servers' bandwidth so that the available bandwidth per simulated host remained constant (35 Mbps). For the PBFT simulation, the 1 MB block was too big to handle, thus the PBFT line corresponds to a 250 KB block size.

As Figure 3.5 shows, the simple version of ByzCoin achieves acceptable latency, as long as the consensus group size is less than 200. After this point the cost for the leader to broadcast the block to everyone incurs large overheads. On the contrary, the tree-based ByzCoin scales well, since the same 1 MB block for 1008 nodes suffers signing latency less than the flat approach for 36 nodes. Adding 28 times more nodes (from 36 to 1008) causes a latency increase close to a factor 2 (from 6.5 to 14 seconds). The basic PBFT implementation is quite fast for 2 nodes but scales poorly (40 seconds for 100 nodes), whereas the tree-based implementation with individual signatures performs the same as ByzCoin for up to 200 hosts. If we aim for the higher security level of 1008 nodes, however, then ByzCoin is 3 times faster.

Figure 3.6 shows the performance cost of keyblock signing. The flat variant outperforms the tree-based version when the number of hosts is small since the blocks have as many transactions as there are hosts and thus are small themselves. This leads to a fast transmission even in the flat case and the main overhead comes from the block propagation latency, which scales with $O(\log N)$ in the tree-based ByzCoin variant.

Block Size Comparison

The next experiment analyzes how different block sizes affect the scalability of ByzCoin. We used a constant number of 144 hosts for all implementations. Once again, PBFT was unable to achieve acceptable latency with 144 nodes, thus we ran it with 100 nodes only.

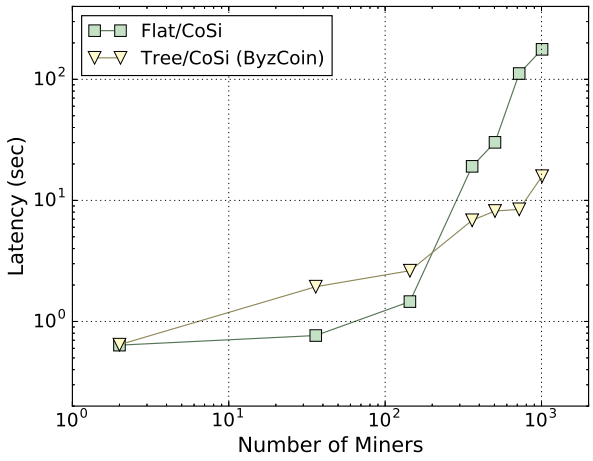

Figure 3.6 – Keyblock signing latency

Figure 3.7 shows the average latency of the consensus mechanism, determined over 10 blocks when their respective sizes increase. As in the previous section we see that the flat implementation is acceptable for a 1 MB block, but when the block increases to 2 MB the latency quadruples. This outcome is expected as the leader's link saturates when he tries to send 2 MB messages to every participating node. In contrast ByzCoin scales well because the leader outsources the transmission of the blocks to other nodes and contacts only his children. The same behavior is observed for the algorithm that uses individual signatures and tree-based communication, which shows that the block size has no negative effect on scalability when a tree is used. Finally, we find that PBFT is fast for small blocks, but the latency rapidly increases to 40 seconds for 250 KB blocks.

ByzCoin's signing latency for a 1 MB block is close to 10 seconds, which should be small enough to make the need for 0-confirmation transactions almost disappear. Even for a 32 MB block ($\approx$ 66000 transactions) the delay is much lower (around 90 seconds) than the $\approx$ 10 minutes required by Bitcoin.

Figure 3.8 demonstrates the signing latency of various blocks sizes on tree-based ByzCoin. Signing one-transaction blocks takes only 3 seconds even when 1008 miners co-sign it. For bigger blocks, we have included Bitcoin's current maximum block size of 1 MB along with the proposed limits of 2 MB in Bitcoin Classic and 8 MB in Bitcoin Unlimited [9]. As the graph shows, 1 MB and 2 MB blocks scale linearly in the number of nodes at first but after 200 nodes, the propagation latency is higher than the transmission of the block, hence the latency is close to constant. For 8 MB blocks, even with 1008 the signing latency increases only linearly.

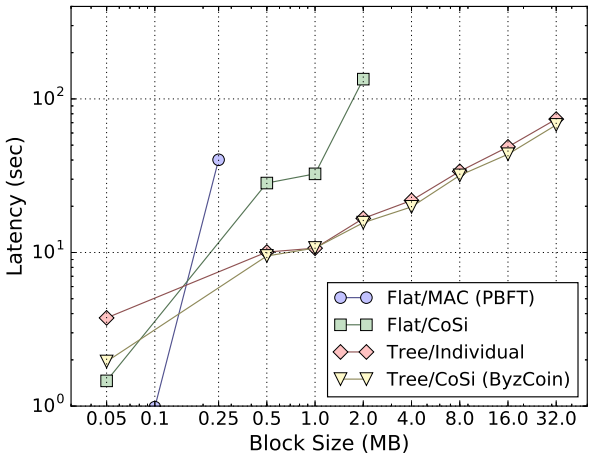

Figure 3.7 – Influence of the block size on the consensus latency

3.3.4 Transaction Throughput

In this experiment, we investigate the maximum throughput in terms of transactions per second (TPS) that ByzCoin can achieve and show how Bitcoin could improve its throughput by adopting a ByzCoin-like deployment model. We tested ByzCoin versions with consensus group sizes of 144 and 1008 nodes, respectively. Note that performance-wise this resembles the worst case scenario since the miner-share ratio is usually not 1:1 as miners in the consensus group are allowed to hold multiple shares, as described in Section 3.2.3.

Analyzing Figure 3.9 shows that Bitcoin can increase its overall throughput by more than one order of magnitude through the adoption of a flat ByzCoin-like model, which separates transaction verification and block mining and deals with forks via strong consistency. Furthermore, the 144 node configuration can achieve close to 1000 TPS, which is double the throughput of Paypal, and even the 1008-node roster achieves close to 700 TPS. Even when the tree fails, the system can revert back to 1 MB microblocks on the flat and more robust variant of ByzCoin and still have a throughput of ten times higher than the current maximum throughput of Bitcoin.

In both Figure 3.7 and Figure 3.9, the usual trade-off between throughput and latency appears. The system can work with 1–2 MB microblocks when the load is normal and then has a latency of 10–20 seconds. If an overload occurs, the system adaptively changes the block size to enable higher throughput. We note that this is not the case in the simple ByzCoin where 1 MB microblocks have optimal throughput and acceptable latency.

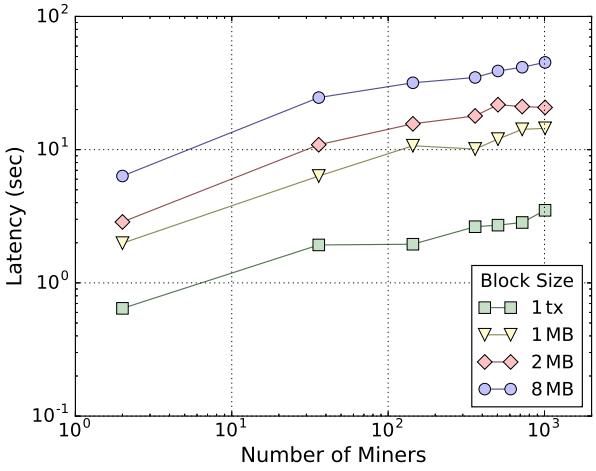

Figure 3.8 – Influence of the consensus group size on the block signing latency

3.4 Security Analysis

In this section, we conduct a preliminary, informal security analysis of ByzCoin, and discuss how its consensus mechanism can mitigate or eliminate some known attacks against Bitcoin.

3.4.1 Transaction Safety

In the original Bitcoin paper [191], Nakamoto models Bitcoin's security against transaction double spending attacks as in a Gambler's Ruin Problem. Furthermore, he models the progress an attacker can make as a Poisson distribution and combines these two models to reach Equation 3.1. This equation calculates the probability of a successful double spend after z blocks when the adversary controls q computing power.

$$P = 1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - \left(\frac{q}{p}\right)^{(z-k)}\right) \tag{3.1}$$

In Figure 3.10 and Figure 3.11 we compare the relative safety of a transaction over time in Bitcoin[2] versus ByzCoin. Figure 3.10 shows that ByzCoin can secure a transaction in less than a minute, because the collective signature guarantees forward security. On the contrary, Bitcoin's transactions need hours to be considered fully secured from a double-spending attempt. Figure 3.11 illustrates the required time from transaction creation to the point where a double spending attack has less than 0.1% chance of success. ByzCoin incurs a latency of below one minute to achieve the above security, which boils down to the time the systems needs to produce a collectively signed microblock. Bitcoin, on the other hand, needs

[2] Based on data from https://blockchain.info.

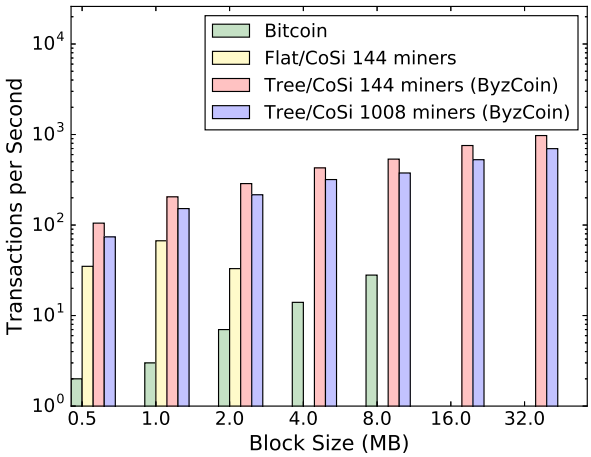

Figure 3.9 – Throughput of ByzCoin

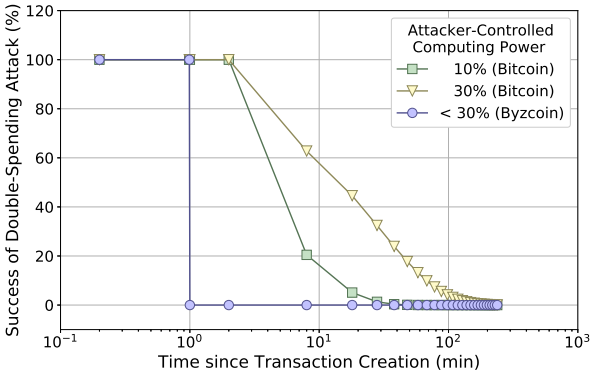

Figure 3.10 – Successful double-spending attack probability

several hours to reach the same guarantees. Note that this graph does not consider other advanced attacks, such as eclipse attacks [128], where Bitcoin offers no security for the victim's transactions.

3.4.2 Proof-of-Membership Security

The security of ByzCoin's proof-of-membership mechanism can be modeled as a random sampling problem with two possible independent outcomes (honest, Byzantine). The probability of picking a Byzantine node (in the worst case) is $p = 0.25$ and the number of tries corresponds to the share window size w. In this setting, we are interested in the probability that the system picks less than $c = \lfloor \frac{w}{3} \rfloor$ Byzantine nodes as consensus group members and hence guarantees safety. To calculate this probability, we use the cumulative binomial distribution where X is the random variable that represents the number of times we pick a Byzantine

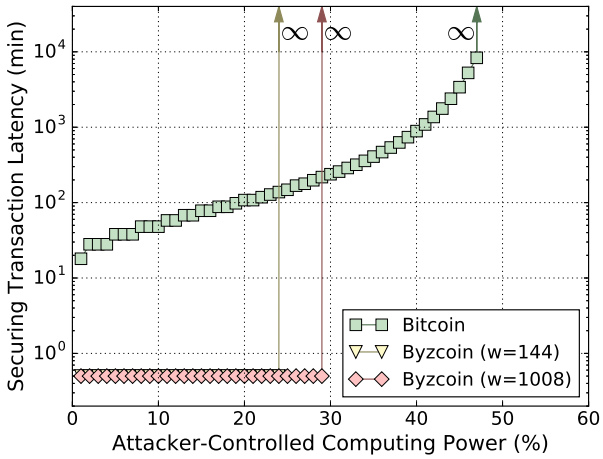

Figure 3.11 – Client-perceived secure transaction latency

node:

$$P[X \leq c] = \sum_{k=0}^{c} \binom{w}{k} p^k (1-p)^{w-k} \tag{3.2}$$

Table 3.2 displays the results for the evaluation of Equation 3.2 for various window sizes w both in the common threat model where an adversary controls up to 25% hash power and in the situation where the system faces a stronger adversary with up to 30% computing power. The latter might temporarily occur due to hash power variations and resource churn.

Table 3.2 – Expected proof-of-membership security levels

$p \mid w$	12	100	144	288	1008	2016
0.25	0.842	0.972	0.990	0.999	0.999	1.000
0.30	0.723	0.779	0.832	0.902	0.989	0.999

At this point, recall that w specifies the number of available shares and not necessarily the number of actual miners as each member of the consensus group is allowed to hold multiple shares. This means that the number of available shares gives an upper bound on the latency of the consensus mechanism with the worst case being that each member holds exactly one share.

In order to choose a value for w appropriately one must take into account not only consensus latency and the desired security level (ideally ≥ 99%) but also the increased chance for resource churn when values of w become large. From a security perspective, the results of Table 3.2 suggest that the share window size should not be set to values lower than $w = 144$. Ideally,

values of $w = 288$ and above should be chosen to obtain a reasonable security margin and, as demonstrated in Section 5.7, values up to $w = 1008$ provide excellent performance numbers.

Finally, care should be taken when bootstrapping the protocol, as for small values of w there is a high probability that a malicious adversary is able to take over control. For this reason, we suggest that BYZCOIN starts with vanilla Nakamoto consensus and only after w keyblocks are mined the BYZCOIN consensus is activated.

3.4.3 Defense Against Bitcoin Attacks

0-confirmation Double-Spend Attacks Race [140] and Finney [101] attacks belong to the family of 0-confirmation double-spend attacks which might affect traders that provide real-time services to their clients. In such scenarios the time between exchange of currency and goods is usually short because traders often cannot afford to wait an extended period of time (10 or more minutes) until a transaction they received can be considered indeed confirmed.

BYZCOIN can mitigate both attacks by putting the merchant's transaction in a collectively signed microblock whose verification latency is in the order of a few seconds up to a minute. If this latency is also unacceptable, then he can send a single transaction for signing, which will cost more, but is secured in less than 4 seconds.

N-confirmation Double-Spend Attacks The assumption underlying this family of attacks [30] is that, after receiving a transaction for a trade, a merchant waits for $N-1$ additional blocks until he concludes the interaction with his client. At this point, a malicious client creates a new double-spending transaction and tries to fork the blockchain, which has a non-negligible success-probability if the adversary has enough hash power. For example, if $N = 3$ then an adversary holding 10% of the network's hash power has a 5% success-chance to mount the above attack [191].

In BYZCOIN, the merchant would simply check the collective signature of the microblock that includes the transaction, which allows him to verify that it was accepted by a super-majority of the network. Afterward, the attacker cannot succeed in forking the blockchain as the rest of the signers will not accept his new block. Even if the attacker is the leader, the proposed microblock will be rejected, and a view change will occur.

Eclipse and Delivery-Tampering Attacks In an eclipse attack [128] it is assumed that an adversary controls a sufficiently large number of connections between the victim and the Bitcoin network. This enables the attacker to mount attacks such as 0- and N-confirmation double-spends with an ever increasing chance of success the longer the adversary manages to keep his control over the network. Delivery-tampering attacks [113] exploit Bitcoin's scalability measures to delay propagation of blocks without causing a network partition. This allows an adversary to control information that the victim receives and simplifies to mount 0- and

1-confirmation double-spend attacks as well as selfish-mining.

While ByzCoin does not prevent an attacker from eclipsing a victim or delaying messages in the peer-to-peer network, its use of collective signatures in transaction commitment ensure that a victim cannot be tricked into accepting an alternate attacker-controlled transaction history produced in a partitioned network fragment.

Selfish and Stubborn Mining Attacks Selfish mining [96] allows a miner to increase his profit by adding newly mined blocks to a hidden blockchain instead of instantly broadcasting them. This effect can be further amplified if the malicious miner has good connectivity to the Bitcoin network. The authors of selfish mining propose a countermeasure that thwarts the attack if a miner has less than 25% hash power under normal circumstances or less than 33% in case of an optimal network. Stubborn mining [194] further generalizes the ideas behind selfish mining and combines it with eclipse attacks in order to increase the adversary's revenue.

In ByzCoin, these strategies are ineffective as (1) forks are instantly resolved in a deterministic manner and (2) the back-link of a key-block is derived by hashing the collectively-signed block header of the previous key-block, which is impossible to generate without the cooperation of honest miners. Hence, building a valid hidden blockchain is impossible.

Transaction Censorship In Bitcoin-NG, a malicious leader can censor transactions for the duration of his epoch(s). The same applies to ByzCoin. However, as not every leader is malicious, the censored transactions are only delayed and will be processed eventually by the next honest leader. ByzCoin can improve on this, as the leader's actions are double-checked by all the other miners in the consensus group. A client can announce his censored transaction just like in classic PBFT; this will indicate a potential leader fault and will either stop censorship efforts or lead to a view-change to remove the malicious leader. Finally, in Bitcoin(-NG) a miner can announce his intention to fork over a block that includes a transaction, giving an incentive to other miners to exclude this transaction. In ByzCoin using fork-based attacks to censor transactions is no longer possible due to ByzCoin's deterministic fork resolution mechanism. An attacker can therefore only vote down a leader's proposals by refusing to co-sign. This is also a limitation, however, as an adversary who controls more than 33% of the shares (Section 3.5) deny service and can censor transactions for as long as he wants.

3.5 Limitations and Future Work

This section briefly outlines several of ByzCoin's important remaining limitations, and areas for future work.

Consensus-Group Exclusion A malicious ByzCoin leader can potentially exclude nodes from the consensus process. This is easier in the flat variant, where the leader is responsible for contacting every participating miner, but it is also possible in the tree-based version if the leader "reorganizes" the tree and puts nodes targeted for exclusion in subtrees where the roots are colluding nodes. A malicious leader faces a dilemma, though: excluded nodes lose their share of newly minted coins which increases the overall value per coin and thus the leader's reward. The victims, however, will quickly broadcast view-change messages in an attempt to remove the Byzantine leader.

As an additional countermeasure to mitigate such an attack, miners could run a peer-to-peer network on top of the tree to communicate protocol details. Thus each node potentially receives information from multiple sources. If the parent of a node fails to deliver the announcement message of a new round, this node could then choose to attach itself (together with its entire subtree) to another participating (honest) miner. This self-adapting tree could mitigate the leader's effort to exclude miners. As a last resort, the malicious leader could exclude the commitments of the victims from the aggregate commitment, but as parts of the tree have witnessed these commitments, the risk of triggering a view-change is high.

In summary, the above attack seems irrational as the drawbacks of trying to exclude miners seem to outweigh the benefits. We leave a more thorough analysis of this situation for future work.

Defenses Against 33%+ Attacks An attacker powerful enough to control more than $\frac{1}{3}$ of the consensus shares can, in the Byzantine threat model, trivially censor transactions by withholding votes, and double-spend by splitting honest nodes into two disjoint groups and collecting enough signatures for two conflicting microblocks. Figure 3.11 shows how the safety of ByzCoin fails at 30%, whereas Bitcoin remains safe even for 48%, if a client can wait long enough.

However, the assumption that an attacker completely controls the network is rather unrealistic, especially if messages are authenticated and spoofing is impossible [13]. The existence of the peer-to-peer network on top of the tree, mentioned in the previous paragraph, enables the detection of equivocation attacks such as microblock forks and mitigates the double-spending efforts, as honest nodes will stop following the leader. Thus, double-spending and history rewriting attacks in ByzCoin become trivial only after the attacker has 66% of the shares, effectively increasing the threshold from 51% to 66%. This assumption is realistic, as an attacker controlling the complete network can actually split Bitcoin's network into two halves and trivially double-spend on the weaker side. This is possible because the weak side creates blocks that will be orphaned once the partition heals. We again leave a more thorough analysis of this situation for future work.

Proof-of-Work Alternatives Bitcoin's hash-based proof-of-work has many drawbacks, such as energy waste and the efficiency advantages of custom ASICs that have made mining by "normal users" impractical. Many promising alternatives are available, such as memory-intensive puzzles [14], or proof-of-stake designs [147]. Consensus group membership might in principle also be based on other Sybil attack-resistant methods, such as those based on social trust networks [254, 151] or Proof-of-Personhood [43]. A more democratic alternative might be to apportion mining power on a "1 person, 1 vote" principle, based on anonymous *proof-of-personhood* tokens distributed at pseudonym parties [105]. Regardless, we treat the ideal choice of Sybil attack-resistance mechanism as an issue for future work, orthogonal to the focus of this thesis.

Other Directions Besides the issues outlined above, there are many more interesting open questions worth considering: Sharding [70] presents a promising approach to scale distributed protocols and was already studied for private blockchains [74]. A sharded variant of ByzCoin might thus achieve even better scalability and performance numbers. A key obstacle that needs to be analyzed in that context before though is the generation of bias-resistant public randomness [165] which would enable to pick members of a shard in a distributed and secure manner. Yet another challenge is to find ways to increase incentives of rational miners to remain honest, like binding coins and destroying them when misbehavior is detected [46]. Finally, asynchronous BFT [49, 48] is another interesting class of protocols, which only recently started to be analyzed in the context of blockchains [185].

3.6 Conclusion

ByzCoin is a scalable Byzantine fault tolerant consensus algorithm for open decentralized blockchain systems such as Bitcoin. ByzCoin's strong consistency increases Bitcoin's core security guarantees—shielding against attacks on the consensus and mining system such as N-confirmation double-spending, intentional blockchain forks, and selfish mining—and also enables high scalability and low transaction latency. ByzCoin's application to Bitcoin is just one example, though: theoretically, it can be deployed to any blockchain-based system and the proof-of-work-based leader election mechanism might be changed to another approach such as proof-of-stake. If open membership is not an objective, the consensus group could be static, though still potentially large. We developed a wide-scale prototype implementation of ByzCoin, validated its efficiency with measurements and experiments, and have shown that Bitcoin can increase the capacity of transactions it handles by more than two orders of magnitude.

4 Scalable Bias-Resistant Distributed Randomness

4.1 Introduction

A reliable source of randomness that provides high-entropy output is a critical component in many protocols [37, 68]. The reliability of the source, however, is often not the only criterion that matters. In many high-stakes protocols, the unbiasability and public-verifiability of the randomness generation process are as important as ensuring that the produced randomness is good in terms of the entropy it provides [115].

More concretely, Tor hidden services [82] depend on the generation of a fresh random value each day for protection against popularity estimations and DoS attacks [119]. Anytrust-based systems, such as Herbivore [117], Dissent [249], and Vuvuzela [245], as well as sharded blockchains [70], use bias-resistant public randomness for scalability by sharding participants into smaller groups. TorPath [114] critically depends on public randomness for setting up the consensus groups. Public randomness can be used to transparently select parameters for cryptographic protocols or standards, such as in the generation of elliptic curves [18, 165], where adversaries should not be able to steer the process to select curves with weak security parameters [26]. Other use-cases for public randomness include voting systems [4] for sampling ballots for manual recounts, lotteries for choosing winning numbers, and Byzantine agreement algorithms [49, 196] for achieving scalability.

The process of generating public randomness is nontrivial, because obtaining access to sources of good randomness, even in terms of entropy alone, is often difficult and error-prone [58, 123]. One approach is to rely on randomness beacons, which were introduced by Rabin [210] in the context of contract signing, where a trusted third party regularly emits randomly chosen integers to the public. The NIST beacon [193] provides hardware-generated random output from quantum-mechanical effects, but it requires trust in their centralized beacon—a problematic assumption, especially after the Dual EC DRBG debacle [29, 230].

This work is concerned primarily with the trustworthiness, rather than the entropy, of public randomness sources. Generating public randomness without a trusted party is often desirable,

especially in decentralized settings such as blockchains, where many mutually-distrustful users may wish to participate. Producing and using randomness in a distributed setting presents many issues and challenges, however, such as how to choose a subset of available beacons, or how to combine random outputs from multiple beacons without permitting bias by an active adversary. Prior approaches to randomness without trusted parties [206] employ Bitcoin [41], slow cryptographic hash functions [165], lotteries [18], or financial data [60] as sources for public randomness.

Our goal is to provide bias-resistant public randomness in the familiar (t, n)-threshold security model already widely-used both in threshold cryptography [202] and Byzantine consensus protocols [49]. Generating public randomness is hard, however, as active adversaries can behave dishonestly in order to bias public random choices toward their advantage, *e.g.,* by manipulating their own explicit inputs or by selectively injecting failures. Although addressing those issues is relatively straightforward for small values of $n \approx 10$ [49, 141], we address scalability challenges of using larger values of n, in the hundreds or thousands, for enhanced security in real-world scenarios. For example, scalable randomness is relevant for public cryptocurrencies [149, 191] which tend to have hundreds to thousands of distinct miners or for countries with thousands of national banks that might want to form a national permissioned blockchain with secure random sharding.

This work's contributions are mainly pragmatic rather than theoretical, building on existing cryptographic primitives to produce more scalable and efficient distributed randomness protocols. We introduce two scalable public-randomness generation protocols: RANDHOUND is a "one-shot" protocol to generate a single random output on demand, while RANDHERD is a randomness beacon protocol that produces a regular series of random outputs. Both protocols provide the same key security properties of unbiasability, unpredictability, availability, and third-party verifiability of their random outputs.

RANDHOUND is a client-server randomness scavenging protocol enabling a client to gather fresh randomness on demand from a potentially large set of nearly-stateless randomness servers, preferably run by independent parties. A party that occasionally requires trustworthy public randomness, such as a lottery association, can use RandHound to produce a random output that includes contributions of – and trustworthiness attestations from – all participating servers. The RandHound client (*e.g.,* the lottery association) first publicly commits to the parameters of a unique RandHound protocol run, such as the time and date of the lottery and the set of servers involved, so a malicious client cannot bias the result by secretly rerunning the protocol. The client then splits the servers into balanced subgroups for scalability. Each subgroup uses publicly verifiable secret sharing (PVSS) [222, 234] to produce secret inputs such that an honest threshold of participants can later recover them and form a third-party-verifiable proof of their validity. To tolerate server failures, the client selects a subset of secret inputs from each group. Application of the pigeonhole principle ensures the integrity of RANDHOUND's final output even if some subgroups are compromised, *e.g.,* due to biased grouping. The client commits to his choice of secrets, to prevent equivocation, by obtaining

a collective signature [243] from participating servers. After the servers release the selected secrets, the client combines and publishes the collective random output along with a third-party verifiable transcript of the protocol run. Anyone can subsequently check this transcript to verify that the random output is trustworthy and unbiased, provided not too many servers were compromised.

RANDHERD is a complementary protocol enabling a potentially large collection of servers to form a distributed public randomness beacon, which proactively generates a regular series of public random outputs. RandHerd runs continually and need not be initiated by any client, but requires stateful servers. No single or sub-threshold group of failing or malicious servers can halt the protocol, or predict or significantly bias its output. Clients can check the trustworthiness of any published beacon output with a single, efficient check of one collective signature [243]. RandHerd first invokes RANDHOUND once, at setup or reconfiguration time, to divide the set of servers securely into uniformly random groups, and to generate a short-term aggregate public key used to produce and verify individual beacon outputs. RANDHERD subsequently uses a threshold collective signing protocol based on Shamir secret sharing [33, 227], to generate random outputs at regular intervals. Each of RANDHERD's random outputs doubles as a collective Schnorr signature [235, 243], which clients can validate efficiently against the group's aggregate public key.

The dominant cost in both protocols is publicly verifiable secret sharing (PVSS), which normally incurs $O(n^3)$ communication and computation costs on each of n participants. RANDHOUND and RANDHERD run PVSS only among smaller groups, however, whose configured size c serves as a security parameter. RANDHOUND, therefore, reduces the asymptotic cost to $O(n)$ if c is constant. By leveraging efficient tree-structured communication, RANDHERD further reduces the cost of producing successive beacon outputs to $O(\log n)$ per server.

We implemented the RANDHOUND and RANDHERD protocols in Go, and made these implementations freely available as part of the EPFL DEDIS lab's Cothority framework.[1] Experiments with our prototype implementations show that, among a collective of 512 globally-distributed servers divided into groups of 32, RANDHERD can produce a new 32-byte collective random output every 6 seconds, following a one-time setup process using RANDHOUND that takes approximately 260 seconds. The randomness verification overhead of RANDHERD is equivalent to verifying a single Schnorr multisignature [221], typically less than 100 bytes in size, which clients can check in constant time. Using RANDHOUND alone to produce a random output on demand, it takes approximately 240 seconds to produce randomness and approximately 76 seconds to verify it using the produced 4 MByte transcript. In this configuration, a Byzantine adversary can compromise the availability of either protocol with a probability of at most 0.08%.

Contributions. In this work we introduce three protocols to aid public-randomness generation in an unbiasable, and fully distributed way. In particular, we propose:

[1] https://github.com/dedis/cothority

1. RANDSHARE: A Small-Scale Unbiasable Randomness Protocol. (Section 4.2.2)
2. RANDHOUND: A Verifiable Randomness Scavenging Protocol. (Section 4.3)
3. RANDHERD: A Scalable Randomness Cothority. (Section 4.4)

All three protocols provide three main properties: *unbiasability*, *unpredictability*, and *availability*. *Unbiasability* ensures that an adversary cannot influence the random output, *unpredictability* prevents the adversary from prematurely learning the output even if he cannot affect it, and *availability* enables honest participants to successfully obtain randomness. RANDSHARE serves as a motivational example on how to achieve these properties in a scenario when there is a small number of highly-available participants. RANDHOUND and RANDHERD add *third-party verifiability* to the randomness they produce. RANDHOUND is a client/server protocol that enables the client to obtain randomness with the cooperation of a potentially large set of servers. Finally, RANDHERD balances security and performance trade-offs by using RANDHOUND to set up a large-scale randomness cothority that efficiently produces randomness at frequent intervals. Further, we provide proof-of-concept prototypes for RANDHOUND and RANDHERD to demonstrate that we can achieve these properties as well as good performance.

This chapter is organized as follows. First, Section 4.2 introduces some naive approaches to generate distributed randomness and a correct but non-scalable way. Then, Sections 4.3 and 4.4 introduce the design and security properties of RANDHOUND and RANDHERD, respectively. Finally, Section 4.5 evaluates the prototype implementations of both protocols and Section 4.6 concludes.

4.2 How (not) to Generate Randomness

We first introduce notation which RANDHOUND and RANDHERD build on. We then consider a series of strawman protocols illustrating the key challenges in distributed randomness generation of commitment, selective aborts, and malicious secret shares. We end with RANDSHARE, a protocol that offers the desired properties, but unlike RANDHOUND and RANDHERD is not third-party verifiable and does not scale well.

For the rest of the work, we denote by $\mathcal{G}$ a multiplicatively written cyclic group of order q with generator G, where the set of non-identity elements in $\mathcal{G}$ is written as $\mathcal{G}^*$. We denote by $(x_i)_{i\in I}$ a vector of length $|I|$ with elements x_i, for $i \in I$. Unless stated otherwise, we denote the private key of a node i by x_i and the corresponding public key by $X_i = G^{x_i}$.

4.2.1 Insecure Approaches to Public Randomness

For expositional clarity, we summarize a series of inadequate strawman designs: (I) a naive, trivially insecure design, (II) one that uses a commit-then-reveal process to ensure unpre-

dictability but fails to be unbiasable, and (III) one that uses secret sharing to ensure unbiasability in an honest-but-curious setting, but is breakable by malicious participants.

Strawman I. The simplest protocol for producing a random output $r = \bigoplus_{i=0}^{n-1} r_i$ requires each peer i to contribute their secret input r_i under the (false) assumption that a random input from any honest peer would ensure unbiasability of r. However, a dishonest peer j can force the output value to be $\hat{r}$ by choosing $r_j = \hat{r} \bigoplus_{i:i\neq j} r_i$ upon seeing all other inputs.

Strawman II. To prevent the above attack, we want to force each peer to commit to their chosen input *before* seeing other inputs by using a simple *commit-then-reveal* approach. Although the output becomes unpredictable as it is fixed during the commitment phase, it is not unbiasable because a dishonest peer can choose not to reveal his input upon seeing all other openings of committed inputs. By repeatedly forcing the protocol to restart, the dishonest peer can obtain an output that is beneficial for him, even though he cannot choose its exact value. The above scenario shows an important yet subtle difference between an output that is *unbiased* when a single, successful run of the protocol is considered, and an output that is *unbiasable* in a more realistic scenario, when the protocol repeats until some output is produced. An attacker's ability to re-toss otherwise-random coins he does not like is central to the reason peer-to-peer networks that use cryptographic hashes as participant IDs are vulnerable to clustering attacks [166].

Strawman III. To address this issue, we wish to ensure that a dishonest peer either cannot force the protocol to abort by refusing to participate, or cannot benefit from doing so. Using a (t, n)-secret sharing scheme, we can force the adversary to commit to his action *before* knowing which action is favorable to him. First, all n peers, where at most f are dishonest, distribute secret shares of their inputs using a $t = f + 1$ recovery threshold. Only after each peer receives n shares will they reconstruct their inputs and generate r. The threshold $t = f + 1$ prevents a dishonest peer from learning anything about the output value. Therefore, he must blindly choose to abort the protocol or to distribute his share. Honest peers can then complete the protocol even if he stops participating upon seeing the recovered inputs. Unfortunately, a dishonest peer can still misbehave by producing bad shares, preventing honest peers from successfully recovering identical secrets.

4.2.2 RandShare: Small-Scale Unbiasable Randomness Protocol

RANDSHARE is an unbiasable randomness protocol that ensures unbiasability , unpredictability, and availability, but is practical only at small scale due to $O(n^3)$ communication overhead. RANDSHARE introduces key concepts that we will re-use in the more scalable RANDHOUND protocol (Section 4.3).

RANDSHARE extends the approach for distributed key-generation in a synchronous model of Gennaro et al. [111] by adopting a point-of-no-return strategy implemented through the concept of a *barrier*, a specific point in the protocol execution after which the protocol always

completes successfully, and by extending it to the asynchronous setting, where the adversary can break timing assumptions [48, 49].

In RANDSHARE, the protocol output is unknown but fixed as a function of $f+1$ inputs. After the barrier point, the protocol output cannot be changed and all honest peers eventually output the previously fixed value, regardless of the adversary's behavior. In RANDSHARE, we define the barrier at the point where the first honest member reveals the shares he holds.

We assume a Byzantine adversary and an asynchronous network where messages are eventually delivered. Let $\mathrm{N} = \{1, \ldots, n\}$ denote the list of peers that participate in RANDSHARE and $n = 3f+1$, where f is the number of dishonest peers. Let $t = f+1$ be the VSS threshold. We assume every peer has a copy of a public key X_j for all $j \neq i$, and that only valid, properly-signed messages are accepted.

Each RANDSHARE peer $i \in \mathrm{N}$ executes the following steps:

1. Share Distribution.

1. Select coefficients $a_{ik} \in_R \mathbb{Z}_q^*$ of a degree $t-1$ secret sharing polynomial $s_i(x) = \sum_{k=0}^{t-1} a_{ik} x^k$. The secret to be shared is $s_i(0) = a_{i0}$.
2. Compute polynomial commitments $A_{ik} = G^{a_{ik}}$, for all $k \in \{0, \ldots, t-1\}$, and calculate secret shares $s_i(j)$ for all $j \in \mathrm{N}$.
3. Securely send $s_i(j)$ to peer $j \neq i$ and start a Byzantine agreement (BA) run on $s_i(0)$, by broadcasting $\widehat{A}_i = (A_{ik})_{k \in \{0,\ldots,t-1\}}$.

2. Share Verification.

1. Initialize a bit-vector $V_i = (v_{i1}, \ldots, v_{in})$ to zero, to keep track of valid secrets $s_j(0)$ received. Then wait until a message with share $s_j(i)$ from each $j \neq i$ has arrived.
2. Verify that each $s_j(i)$ is valid using $\widehat{A}_j$. This may be done by checking that $S_j(i) = G^{s_j(i)}$ where:
$$S_j(x) = \prod_{k=0}^{t-1} A_{jk}^{x^k} = G^{\sum_{k=0}^{t-1} a_{jk} x^k} = G^{s_j(x)}$$
3. If verification succeeds, confirm $s_j(i)$ by broadcasting the prepare message $(\mathsf{p}, i, j, 1)$ as a positive vote on the BA instance of $s_j(0)$. Otherwise, broadcast $(\mathsf{p}, i, j, s_j(i))$ as a negative vote. This also includes the scenario when $\widehat{A}_j$ was never received.
4. If there are at least $2f+1$ positive votes for secret $s_j(0)$, broadcast $(\mathsf{c}, i, j, 1)$ as a positive commitment. If there are at least $f+1$ negative votes for secret $s_j(0)$, broadcast $(\mathsf{c}, i, j, 0)$ as a negative commitment.

5. If there are at least $2f+1$ commits (c, i, j, x) for secret $s_j(0)$, set $v_{ij} = x$. If $x = 1$, consider the secret recoverable else consider secret $s_j(0)$ invalid.

3. Share Disclosure.

1. Wait until a decision has been taken for all entries of V_i and determine the number of 1-entries n' in V_i.
2. If $n' > f$, broadcast for each 1-entry j in V_i the share $s_j(i)$ and abort otherwise.

4. Randomness Recovery.

1. Wait until at least t shares for each $j \neq i$ have arrived, recover the secret sharing polynomial $s_j(x)$ through Lagrange interpolation, and compute the secret $s_j(0)$.
2. Compute and publish the collective random string as:

$$Z = \bigoplus_{j=1}^{n'} s_j(0)$$

RANDSHARE achieves *unbiasability*, because the secret sharing threshold $t = f+1$ prevents dishonest peers from recovering the honest peers' secrets before the barrier. The Byzantine agreement procedures ensure that all honest peers have a consistent copy of V_i and therefore know which $n' > f$ secrets will be recovered after the barrier or if the protocol run has already failed as $n' \leq f$. Furthermore, if at least $f+1$ honest members sent a success message for each share, and thus Byzantine agreement (with at least $2f+1$ prepares) has been achieved on the validity of these shares, each honest peer will be able to recover *every* other peer's secret value. *Unpredictability* follows from the fact that the final random string Z contains $n' \geq f+1$ secrets; there are at most f malicious peers, and no honest peer will release his shares before the barrier. *Availability* is ensured because $f+1$ honest nodes out of the total $2f+1$ positive voters are able to recover the secrets, given the secret-sharing threshold $t = f+1$, without the collaboration of the dishonest nodes.

4.3 RandHound: Scalable, Verifiable Randomness Scavenging

This section presents RANDHOUND, a scalable client/server protocol for producing public, verifiable, unbiasable randomness. RANDHOUND enables a client, who initiates the protocol, to "scavenge" public randomness from an arbitrary collection of servers. RANDHOUND uses a commit-then-reveal approach to generate randomness, implemented via publicly verifiable secret sharing (PVSS) [222], and it uses CoSi [243] as a witnessing mechanism to fix the protocol output and prevent client equivocation. We first provide an overview of RANDHOUND

and introduce the notation and threat model. We then describe randomness generation and verification in detail, analyze the protocol's security properties, and discuss protocol extensions.

4.3.1 Protocol Overview

RANDHOUND employs a client/server model, in which a client invokes the services of a set of RandHound servers to produce a random value. RANDHOUND assumes the same threat model as RandShare, *i.e.,* that at most f out of at least $3f+1$ participants are dishonest. If the client is honest, we allow at most f servers to be malicious and if the adversary controls the client then we allow at most $f-1$ malicious servers. We assume that dishonest participants can send different but correctly signed messages to honest participants in stages where they are supposed to broadcast the same message to all. Furthermore, we assume that the goal of the adversary is to bias or DoS-attack the protocol run in the honest-client scenario, and to bias the output in the malicious-client scenario.

We assume the client gets only one attempt to run RANDHOUND. A dishonest client might try to run the protocol many times until he obtains a favorable output. However, each protocol run uses a session configuration file C that uniquely identifies a protocol run and binds it to the intended purpose of the random output. To illustrate RANDHOUND's deployment model, the client might be a lottery authority, which must commit ahead of time to all lottery parameters including the time and date of the lottery. A cryptographic hash of the configuration parameters in C uniquely identifies the RANDHOUND protocol instance. If that protocol run fails to produce an output, this failure triggers an alarm and an investigation, and not a silent re-run of the protocol.

Honest RandHound servers enforce this "one-shot" rule by remembering and refusing to participate in a second protocol run with session configuration C until the time-window defined by C has passed. This memory of having recently participated in a session for configuration C is the only state RandHound servers need to store for a significant time; the servers are otherwise largely stateless.

RANDHOUND improves on RANDSHARE's lack of scalability by sharing secrets not directly among all other servers but only within smaller groups of servers. RANDHOUND servers share their secrets only with their respective group members, decreasing the number of shares they create and transmit. This reduces the communication and computational overhead from $O(n^3)$ to $O(nc^2)$, where c is the average (constant) size of a group. The client arranges the servers into disjoint groups. The protocol remains secure even if the client chooses a non-random adversarial grouping, however, because the client must employ all groups and the pigeonhole principle ensures that at least one group is secure.

Each server chooses its random input value and creates shares only for other members of the same group using PVSS. The server sends the encrypted shares to the client together

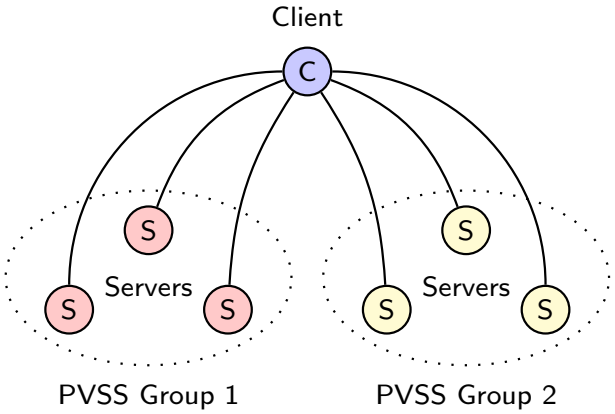

Figure 4.1 – An overview of the RANDHOUND design.

with the NIZK proofs. The client chooses a subset of server inputs from each group, omitting servers that did not respond on time or with proper values, thus fixing each group's secret and consequently the output of the protocol. After the client receives a sign-off on his choice of inputs in a global run of CoSi, the servers decrypt and send their shares to the client. The client, in turn, combines the recovered group secrets to produce the final random output Z. The client documents the run of the protocol in a log L, or *transcript*, by recording the messages he sends and receives. The transcript serves as a third party verifiable proof of the produced randomness. Figure 4.1 gives an overview on the RANDHOUND design.

4.3.2 Description

Let $\mathscr{G}$ be a group of large prime order q with generator G. Let $\mathrm{N} = \{0,\ldots,n-1\}$ denote the list of nodes, let $\mathrm{S} = \mathrm{N} \setminus \{0\}$ denote the list of servers and let f be the maximum number of permitted Byzantine nodes. We require that $n = 3f+1$. We set (x_0, X_0) as the key pair of the client and (x_i, X_i) as the one of server $i > 0$. Further let $T_l \subset \mathrm{S}$, with $l \in \{0,\ldots,m-1\}$, be pairwise disjoint trustee groups and let $t_l = \lfloor |T_l|/3 \rfloor + 1$ be the secret sharing threshold for group T_l.

The publicly available session configuration is denoted by $C = (X, T, f, u, w)$, where $X = (X_0,\ldots,X_{n-1})$ is the list of public keys, $T = (T_0,\ldots,T_{m-1})$ is the server grouping, u is a purpose string, and w is a timestamp. We call $\mathsf{H}(C)$ the session identifier. The session configuration and consequently the session identifier have to be unique for each protocol run. We assume that all nodes know the list of public keys X.

The output of RANDHOUND is a random string Z which is publicly verifiable through a transcript L.

Randomness Generation

RANDHOUND's randomness-generation protocol has seven steps and requires three round trips between the client and the servers; see Figure 4.2 for an overview. All exchanged messages

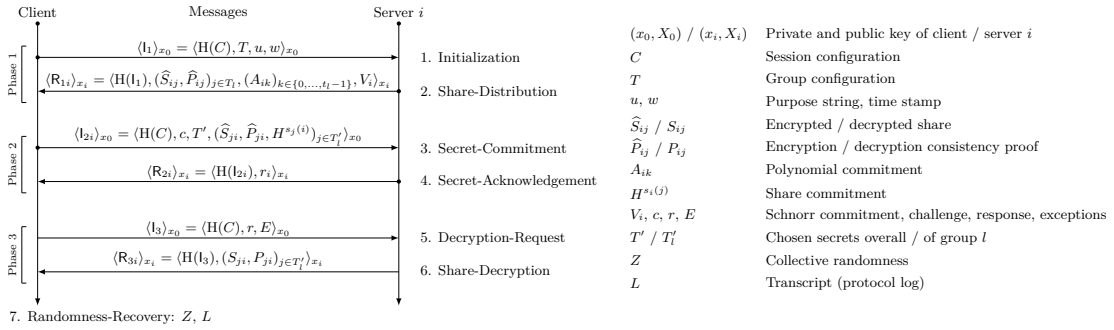

Figure 4.2 – An overview of the RANDHOUND randomness generation process

are signed by the sending party, messages from the client to servers include the session identifier, and messages from servers to the client contain a reply identifier that is the hash of the previous client message. We implicitly assume that client and servers always verify message signatures and session and reply identifiers and that they mark non-authentic or replayed messages and ignore them from the rest of the protocol run.

RANDHOUND consists of three *inquiry-response* phases between the client and the servers followed by the client's randomness recovery.

1. **Initialization (Client).** The client initializes a protocol run by executing the following steps:

   (a) Set the values in C and choose a random integer $r_T \in_R \mathbb{Z}_q$ as a seed to pseudorandomly create a balanced grouping T of S. Record C in L.

   (b) Prepare the message

   $$\langle \mathsf{I}_1 \rangle_{x_0} = \langle \mathsf{H}(C), T, u, w \rangle_{x_0},$$

   record it in L, and broadcast it to all servers.

2. **Share Distribution (Server).** To distribute shares, each trustee $i \in T_l$ executes step 1 of PVSS:

   (a) Map $\mathsf{H}(C)$ to a group element $H \in \mathcal{G}^*$, set $t_l = \lfloor |T_l|/3 \rfloor + 1$, and (randomly) choose a degree $t_l - 1$ secret sharing polynomial $s_i(x)$. The secret to-be-shared is $S_{i0} = G^{s_i(0)}$.

   (b) Create polynomial commitments A_{ik}, for all $k \in \{0, \ldots, t_l - 1\}$, and compute encrypted shares $\widehat{S}_{ij} = X_j^{s_i(j)}$ and consistency proofs $\widehat{P}_{ij}$ for all $j \in T_l$.

   (c) Choose $v_i \in_R \mathbb{Z}_q$ and compute $V_i = G^{v_i}$ as a Schnorr commitment.

   (d) Prepare the message

   $$\langle \mathsf{R}_{1i} \rangle_{x_i} = \langle \mathsf{H}(\mathsf{I}_1), (\widehat{S}_{ij}, \widehat{P}_{ij})_{j \in T_l}, (A_{ik})_{k \in \{0,\ldots,t_l-1\}}, V_i \rangle_{x_i}$$

   and send it back to the client.

3. **Secret Commitment (Client).** The client commits to the set of shared secrets that contribute to the final random string, and asks servers to co-sign his choice:

   (a) Record each received $\langle \mathsf{R}_{1i} \rangle_{x_i}$ message in L.

   (b) Verify all $\widehat{S}_{ij}$ against $\widehat{P}_{ij}$ using X_i and A_{ik}. Buffer each (correct) $H^{s_i(j)}$ created in the process. Mark each share that does not pass the verification as invalid, and do not forward the corresponding tuple $(\widehat{S}_{ij}, \widehat{P}_{ij}, H^{s_i(j)})$ to the respective trustee.

   (c) Create the commitment to the final list of secrets $T' = (T'_0, \ldots, T'_{m-1})$ by randomly selecting $T'_l \subset T_l$ such that $|T'_l| = t_l$ for all $l \in \{0, \ldots, m-1\}$.

   (d) Compute the aggregate Schnorr commit $V = \prod_i V_i$ and the Schnorr challenge $c = \mathsf{H}(V \parallel \mathsf{H}(C) \parallel T')$.

   (e) Prepare the message

   $$\langle \mathsf{I}_{2i} \rangle_{x_0} = \langle \mathsf{H}(C), c, T', (\widehat{S}_{ji}, \widehat{P}_{ji}, H^{s_j(i)})_{j \in T'_l} \rangle_{x_0},$$

   record it in L, and send it to trustee $i \in T_l$.

4. **Secret Acknowledgment (Server).** Each trustee $i \in T_l$ acknowledges the client's commitment by executing the following steps:

   (a) Check that $|T'_l| = t_l$ for each T'_l in T' and that $f + 1 \leq \sum_{l=0}^{m-1} t_l$. Abort if any of those conditions does not hold.

   (b) Compute the Schnorr response $r_i = v_i - c x_i$.

   (c) Prepare the message

   $$\langle \mathsf{R}_{2i} \rangle_{x_i} = \langle \mathsf{H}(\mathsf{I}_{2i}), r_i \rangle_{x_i}$$

   and send it back to the client.

5. **Decryption Request (Client).** The client requests the decryption of the secrets from the trustees by presenting a valid Schnorr signature on his commitment:

   (a) Record each received $\langle \mathsf{R}_{2i} \rangle_{x_i}$ message in L.

   (b) Compute the aggregate Schnorr response $r = \sum_i r_i$ and create a list of exceptions E that contains information on missing server commits and/or responses.

   (c) Prepare the message

   $$\langle \mathsf{I}_3 \rangle_{x_0} = \langle \mathsf{H}(C), r, E \rangle_{x_0},$$

   record it in L, and broadcast it to all servers.

6. **Share Decryption (Server).** To decrypt received shares, each trustee $i \in T_l$ performs step 2 of PVSS:

(a) Check that (c, r) forms a valid Schnorr signature on T' taking exceptions recorded in E into account and verify that at least $2f+1$ servers signed. Abort if any of those conditions do not hold.

(b) Check for all $j \in T'_l$ that $\widehat{S}_{ji}$ verifies against $\widehat{P}_{ji}$ using $H^{s_j(i)}$ and public key X_i.

(c) If the verification fails, mark $\widehat{S}_{ji}$ as invalid and do not decrypt it. Otherwise, decrypt $\widehat{S}_{ji}$ by computing $S_{ji} = (\widehat{S}_{ji})^{x_i^{-1}} = G^{s_j(i)}$ and create a decryption consistency proof P_{ji}.

(d) Prepare the message

$$\langle \mathsf{R}_{3i} \rangle_{x_i} = \langle \mathsf{H}(\mathsf{I}_3), (S_{ji}, P_{ji})_{j \in T'_l} \rangle_{x_i}$$

and send it back to the client.

7. **Randomness Recovery (Client).** To construct the collective randomness, the client performs step 3 of PVSS:

(a) Record all received $\langle \mathsf{R}_{3i} \rangle_{x_i}$ messages in L.

(b) Check each share S_{ji} against P_{ji} and mark invalid ones.

(c) Use Lagrange interpolation to recover the individual S_{i0} that have enough valid shares S_{ij} and abort if even a single one of the secrets previously committed to in T' cannot be reconstructed.

(d) Compute the collective random value as

$$Z = \prod_{i \in \bigcup T'_l} S_{i0} \,,$$

and publish Z and L.

Randomness Verification

A verifier who wants to check the validity of the collective randomness Z against the transcript

$$L = (C, \langle \mathsf{I}_1 \rangle_{x_0}, \langle \mathsf{R}_{1i} \rangle_{x_i}, \langle \mathsf{I}_{2i} \rangle_{x_0}, \langle \mathsf{R}_{2i} \rangle_{x_i}, \langle \mathsf{I}_3 \rangle_{x_0}, \langle \mathsf{R}_{3i} \rangle_{x_i})$$

has to perform the following steps:

1. Verify the values of arguments included in the session configuration $C = (X, T, f, u, w)$. Specifically, check that $|X| = n = 3f+1$, that groups T_l defined in T are non-overlapping and balanced, that $|X| = \sum_{l=0}^{m-1} |T_l|$, that each group threshold satisfies $t_l = |T_l|/3 + 1$, that u and w match the intended use of Z, and that the hash of C matches $\mathsf{H}(C)$ as recorded in the messages.

2. Verify all signatures of $\langle \mathsf{I}_1 \rangle_{x_0}$, $\langle \mathsf{R}_{1i} \rangle_{x_i}$, $\langle \mathsf{I}_{2i} \rangle_{x_0}$, $\langle \mathsf{R}_{2i} \rangle_{x_i}$ $\langle \mathsf{I}_3 \rangle_{x_0}$, and $\langle \mathsf{R}_{3i} \rangle_{x_i}$. Ignore invalid messages for the rest of the verification.

3. Verify that $\mathsf{H}(\mathsf{I}_1)$ matches the hash recorded in R_{1i}. Repeat for I_{2i} and R_{2i}, and I_3 and R_{3i}. Ignore messages that do not include the correct hash.

4. Check that T' contains at least $f+1$ secrets, that the collective signature on T' is valid and that at least $2f+1$ servers contributed to the signature (taking into account the exceptions in E).

5. Verify each recorded encrypted share $\widehat{S}_{ij}$, whose secret was chosen in T', against the proof $\widehat{P}_{ij}$ using X_i and A_{ik}. Abort if there are not enough shares for any secret chosen in T'.

6. Verify each recorded decrypted share S_{ij} against the proof P_{ij} where the corresponding $\widehat{S}_{ij}$ was found to be valid. Abort if there are not enough shares for any secret chosen in T'.

7. Verify Z by recovering Z' from the recovered individual secrets S_{i0} and by checking that $Z = Z'$. If the values are equal, then the collective randomness Z is valid. Otherwise, reject Z.

4.3.3 Security Properties

RANDHOUND provides the following security properties:

1. **Availability.** For an honest client, the protocol successfully completes and produces the final random output Z with high probability.

2. **Unpredictability.** No party learns anything about the final random output Z, except with negligible probability, until the secret shares are revealed.

3. **Unbiasability.** The final random output Z represents an unbiased, uniformly random value, except with negligible probability.

4. **Verifiability.** The collective randomness Z is third-party verifiable against the transcript L, that serves as an unforgeable attestation that the documented set of participants ran the protocol to produce the one-and-only random output Z, except with negligible probability.

In the discussion below, we assume that each honest node follows the protocol and that all cryptographic primitives RANDHOUND uses, provide their intended security properties. Specifically, the (t,n)-PVSS scheme ensures that a secret can be recovered only by using a minimum of t shares and that the shares do not leak information about the secret.

Availability. Our goal is to ensure that an honest client can successfully complete the protocol, even in the presence of adversarial servers that misbehave arbitrarily, including by refusing

to participate. A dishonest client can always abort the protocol, or simply not run it, so we do not consider a "self-DoS" by the client to be an attack on availability. In the remaining security properties, we can thus restrict our concern to attacks in which a dishonest client might corrupt (*e.g.,* bias) the output without affecting the output's availability.

According to the protocol specification, an honest client randomly assigns (honest and dishonest) nodes to their groups. Therefore, each group's ratio of honest to dishonest nodes will closely resemble the overall ratio of honest to dishonest nodes in the entire set. Given that $n = 3f + 1$, the expected number of nodes in a group T_l is about $3f/m$. The secret-sharing threshold of $t_l = |T_l|/3 + 1 = (3f/m)/3 + 1 = f/m + 1$ enables $2f/m$ honest nodes in each group to recover its group secret without the collaboration of malicious nodes. This ensures availability, with high probability, when the client is honest. Section 4.5.3 analyzes of the failure probability of a RANDHOUND run for different parameter configurations.

Unpredictability. We want to ensure that output Z remains unknown to the adversary until step 7 of the protocol, when honest nodes decrypt and reveal the secret shares they hold.

The random output Z is a function of m group secrets, where each group contributes exactly one secret that depends on t_l inputs from group members. Further, each input is recoverable using PVSS with t_l shares. In order to achieve unpredictability, there must be at least one group secret that remains unknown to the adversary until step 7.

We will show that there exists at least one group for which the adversary cannot prematurely recover the group's secret. An adversary who controls the dishonest client can deviate from the protocol description and arbitrarily assign nodes to groups. Assuming that there are h honest nodes in total and m groups, then by the generalized pigeonhole principle, regardless of how the dishonest client assigns the groups, there will be at least one group which contains at least $\lceil h/m \rceil$ nodes. In other words, there must be at least one group with at least an average number of honest nodes. Therefore, we set the threshold for secret recovery for each group l such that the number of nodes needed to recover the group secret contains at least one honest node, that is, $|T_l| - h/m + 1 = f/m + 1$. In RANDHOUND, we have $n = 3f + 1$ and $t_l = |T_l|/3 + 1 = (3f/m)/3 + 1 = f/m + 1$ as needed.

Consequently, the adversary will control at most $m - 1$ groups and obtain at most $m - 1$ group secrets. Based on the properties of PVSS, and the fact that Z is a function of all m group secrets, the adversary cannot reconstruct Z without the shares held by honest nodes that are only revealed in step 7.

Unbiasability. We want to ensure that an adversary cannot influence the value of the random output Z.

In order to prevent the adversary from controlling the output Z, we need to ensure that there exists at least one group for which the adversary does not control the group's secret. If, for each group, the adversary can prematurely recover honest nodes' inputs to the group secret

and therefore be able to prematurely recover all groups' secrets, then the adversary can try many different valid subsets of the groups' commits to find the one that produces the Z most beneficial to him. If for each group, the adversary can exclude honest nodes from contributing inputs to the group secret, then the adversary has full control over all group secrets, hence Z.

As argued in the discussion of unpredictability, there exists at least one group for which the adversary does not control its group secret. Furthermore, the requirement that the client has to select t_l inputs from each group in his commitment T' ensures that at least $\sum_{l=0}^{m-1} t_l = \sum_{l=0}^{m-1} f/m + 1 = f + m$ inputs contribute to the group secrets, and consequently to the output Z. Combining these two arguments, we know that there is at least one group that is not controlled by the adversary and at least one honest input from that group contributes to Z. As a result, the honest member's input randomizes the group's secret and Z, regardless of the adversary's actions.

Lastly, the condition that at least $2f + 1$ servers must sign off on the client's commitment T' ensures that a malicious client cannot arrange malicious nodes in such a way that would enable him to mount a view-splitting attack. Without that last condition, the adversary could use different arrangements of honest and dishonest inputs that contribute to Z and generate multiple collective random values with valid transcripts from which he could choose and release his preferred one.

Verifiability. In RANDHOUND, only the client obtains the final random output Z. In order for Z to be usable in other contexts and by other parties, any third party must be able to independently verify that Z was properly generated. Therefore, the output of RANDHOUND consists of Z and a transcript L, which serves as third-party verifiable proof of Z. The transcript L must (a) enable the third party to replay the protocol execution and (b) be unforgeable.

L contains all messages sent and received during the protocol execution, as well as the session configuration C. If the verifying party finds C acceptable, specifically the identities of participating servers, he can replay the protocol execution and verify the behavior of the client and the servers, as outlined in Section 4.3.2. After a successful protocol run completes, the only relevant protocol inputs that remain secret are the private keys of the client and the servers. Therefore, any third party on its own can verify L and decide on its validity since the private keys are only used to produce signatures and the signatures are verified using the public keys. For the adversary to forge the transcript (produce a different valid transcript without an actual run of the protocol), he must be in possession of the secret keys of all participant listed in C, violating the assumption that at most f nodes are controlled by the adversary.

Therefore, under the assumption that all cryptographic primitives used in RANDHOUND offer their intended security properties, it is infeasible for any party to produce a valid transcript, except by legitimately running the protocol to completion with the willing participation of the at least $\sum_{l=0}^{m-1} |T'_l|$ servers listed in the client's commitment vector T' (step 3).

Further Considerations. In each protocol run, the group element H is derived from the

session identifier $\mathsf{H}(C)$, which mitigates replay attacks. A malicious server that tries to replay an old message is immediately detected by the client, as the replayed PVSS proofs will not verify against the new H. It is also crucial for RANDHOUND's security that none of the participants knows a logarithm a with $G = H^a$. Otherwise the participant can prematurely recover secret shares since $(H^{s_i(j)})^a = H^{as_i(j)} = G^{s_i(j)} = S_{ij}$, which violates RANDHOUND's unpredictability property and might even enable a malicious node to bias the output. This has to be taken into account when deriving H from $\mathsf{H}(C)$. The naive way to map $\mathsf{H}(C)$ to a scalar a and then set $H = G^a$ is obviously insecure as $G = H^{1/a}$. The Elligator mappings [28] provide a secure option for elliptic curves.

4.3.4 Extensions

Each Lagrange interpolation that the client has to perform to recover a server's secret can be replaced by the evaluation of a hash function as follows: Each server i sends, alongside his encrypted shares, the value $\mathsf{H}(s_i(0))$ as a commitment to the client in step 2. After the client's request to decrypt the shares, each server, whose secret was chosen in T', replies directly with $s_i(0)$. The client checks the received value against the server's commitment and, if valid, integrates it into Z.

Note that the verification of the commitment is necessary, as a malicious server could otherwise just send an arbitrary value as his secret that would be integrated into the collective randomness thereby making it unverifiable against the transcript L. The client can still recover the secret as usual from the decrypted shares with Lagrange interpolation if the above check fails or if the respective server is unavailable.

Finally, SCRAPE [53] provides a new approach to decentralized randomness that builds upon an improved version of PVSS. While this approach is orthogonal to ours, the improved PVSS scheme has a lower verification complexity and can be used to reduce the complexity of RANDHOUND from $O(c^2n)$ to $O(cn)$, making it more scalable.

4.4 RandHerd: A Scalable Randomness Cothority

This section introduces RANDHERD, a protocol that builds a collective authority or *cothority* [243] to produce unbiasable and verifiable randomness. RANDHERD serves as a decentralized randomness beacon [193, 210], efficiently generating a regular stream of random outputs. RANDHERD builds on RANDHOUND, but requires no distinguished client to initiate it, and significantly improves repeat-execution performance.

We first outline RANDHERD, then detail the protocol, analyze its security properties, and explore protocol extensions.

4.4.1 Overview

RANDHERD provides a continually-running decentralized service that can generate publicly verifiable and unbiasable randomness on demand, at regular intervals, or both. RANDHERD's goal is to reduce communication and computational overhead of the randomness generation further from RANDHOUND's $O(c^2 n)$ to $O(c^2 \log n)$ given a group size c. To achieve this, RANDHERD requires a one-time setup phase that securely shards cothority nodes into subgroups, then leverages aggregation and communication trees to generate subsequent random outputs. As before, the random output $\widehat{r}$ of RANDHERD is unbiasable and can be verified, together with the corresponding challenge $\widehat{c}$, as a collective Schnorr signature against RANDHERD's collective public key. Section 4.3 illustrates RANDHERD's design.

RANDHERD's design builds on RANDHOUND, CoSi [243], and a (t, n)-threshold Schnorr signature (TSS) scheme [235] that implements threshold-based witness cosigning (TSS-CoSi).

A cothority configuration C defines a given RANDHERD instance, listing the public keys of participating servers and their collective public key X. The RANDHERD protocol consists of RandHerd-Setup, which performs one-time setup, and RandHerd-Round, which produces successive random outputs.

The setup protocol uses RANDHOUND to select a RANDHERD leader at random and arrange nodes into verifiably unbiased random groups. Each group runs the key generation phase of TSS to establish a public group key $\widehat{X}_l$, such that each group member holds a share of the corresponding private key $\widehat{x}_l$. Each group can issue a collective signature with the cooperation of t_l of nodes. All public group keys contribute to the collective RANDHERD public key $\widehat{X}$, which is endorsed by individual servers in a run of CoSi.

Once operational, to produce each random output, RANDHERD generates a collective Schnorr signature $(\widehat{c}, \widehat{r})$ on some input w using TSS-CoSi and outputs $\widehat{r}$ as randomness. TSS-CoSi modifies CoSi to use threshold secret sharing (TSS) rather than CoSi's usual exception mechanism to handle node failures, as required to ensure bias-resistance despite node failures. All m RANDHERD groups contribute to each output, but each group's contribution requires the participation of only t_l members. Using TSS-CoSi to generate and collectively certify random outputs allows clients to verify any RANDHERD output via a simple Schnorr signature check against public key $\widehat{X}$.

4.4.2 Description

Let $\mathrm{N} = \{0, \ldots, n-1\}$ denote the list of all nodes, and let f denote the maximum number of permitted Byzantine nodes. We assume that $n = 3f + 1$. The private and public key of a node $i \in \mathrm{N}$ is x_i and $X_i = G^{x_i}$, respectively. Let C denote the cothority configuration file listing the public keys of all nodes, the cothority's collective public key $\widehat{X} = \prod_{j=0}^{n-1} \widehat{X}_j$, contact information such as IP address and port number, default group sizes for secret sharing, and a timestamp on when C was created. Each node has a copy of C.

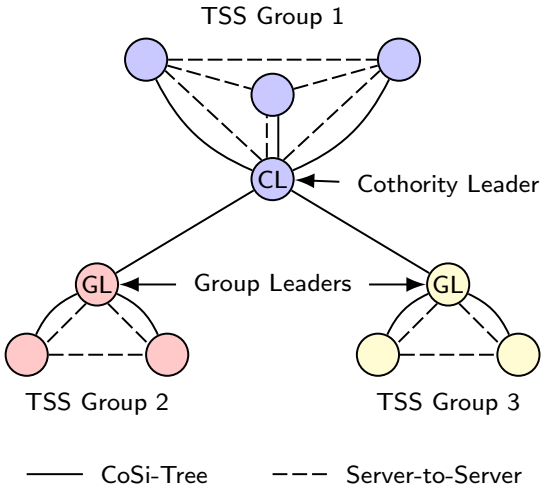

Figure 4.3 – An overview on the RANDHERD design

RandHerd-Setup

The setup phase of RANDHERD consists of the following four steps:

1. **Leader Election.** When RANDHERD starts, each node generates a lottery ticket $t_i = \mathsf{H}(C \parallel X_i)$ for every $i \in \mathrm{N}$ and sorts them in an ascending order. The ticket t_i with the lowest value wins the lottery and the corresponding node i becomes the tentative RANDHERD leader. If this leader is or becomes unavailable, leadership passes to the next node in ascending order. A standard view-change protocol [149, 55] manages the transition between successive leaders. In summary, any server who is dissatisfied with the current leader's progress broadcasts a view-change message for the next leader. Such messages from at least $f+1$ nodes force a view change, and the new leader begins operation upon receiving at least $2f+1$ such "votes of confidence." Section 4.4.5 discusses an improvement to leader election to make successive leaders unpredictable.

2. **Seed Generation.** The leader assumes the role of the RANDHOUND client and runs the protocol, with all other nodes acting as RANDHOUND servers. Each leader has only one chance to complete this step. If he fails, the next node, as determined by the above lottery, steps in and attempts to execute RANDHOUND. After a successful run of RANDHOUND, the leader obtains the tuple (Z, L), where Z is a collective random string and L is the publicly verifiable transcript that proves the validity of Z. Lastly, the current leader broadcasts (Z, L) to all nodes.

3. **Group Setup.** Once the nodes receive (Z, L), they use L to verify Z, and then use Z as a seed to compute a random permutation of N resulting in N′. Afterwards N′ is sharded into m groups T_l of the same size as in RANDHOUND, for $l \in \{0, \ldots, m-1\}$. The node at index 0 of each group becomes the group leader and the group leader of the first group

takes up the role of the temporary RANDHERD leader. If any of the leaders is unavailable, the next one, as specified by the order in N′, steps in. After this step, all nodes know their group assignments and the respective group leaders run a TSS-setup to establish the long-term group secret $\widehat{x}_l$ using a secret sharing threshold of $t_l = |T_l|/3 + 1$. All group leaders report back to the current RANDHERD leader with the public group key $\widehat{X}_l$.

4. **Key Certification.** As soon as the RANDHERD leader has received all $\widehat{X}_j$, he combines them to get the collective RANDHERD public key $\widehat{X} = \prod_{j=0}^{m-1} \widehat{X}_j$ and starts a run of the CoSi protocol to certify $\widehat{X}$ by requesting a signature from each individual node. Therefore, the leader sends $\widehat{X}$ together with all $\widehat{X}_j$ and each individual node checks that $\widehat{X}_j$ corresponds to its public group key and that $\widehat{X}$ is well-formed. Only if both checks succeed, the node participates in the co-signing request, otherwise, it refuses. The collective signature on $\widehat{X}$ is valid if there are least $f/m + 1$ signatures from each group *and* the total number of individual signatures across the groups is at least $2f + 1$. Once a valid signature on $\widehat{X}$ is established, the setup of RANDHERD is completed. The validity of $\widehat{X}$ can be verified by anyone by using the collective public key X, as specified in the configuration C.

After a successful setup, RANDHERD switches to the operational randomness generation mode. Below we describe how the protocol works with an honest and available leader. A dishonest or failed leader can halt progress at any time, but RandHerd-Round uses a view-change protocol as in RandHerd-Setup to recover from leader failures.

RandHerd-Round

In this mode, we distinguish between communications from the RANDHERD leader to group leaders, from group leaders to individual nodes, and communications between all nodes within their respective group. Each randomness generation run consists of the following seven steps and can be executed periodically:

1. **Initialization (Leader).** The RANDHERD leader initializes a protocol run by broadcasting an announcement message containing a timestamp w to all group leaders. All groups will cooperate to produce a signature $(\widehat{c}, \widehat{r})$ on w.

2. **Group Secret Setup / Commitment (Groups / Servers).** Upon the receipt of the announcement, each group creates a short-term secret $\widehat{v}_l$, using a secret sharing threshold t_l, to produce a group commitment $\widehat{V}_l = G^{\widehat{v}_l}$ that will be used towards a signature of w. Furthermore, each individual node randomly chooses $v_i \in_R \mathbb{Z}_q$, creates a commitment $V_i = G^{v_i}$ that will be used to globally witness, hence validate the round challenge $\widehat{c}$, and sends it to the group leader. The group leader aggregates the received individual commitments into $\widetilde{V}_l = \prod_{i \in T_l} V_i$ and sends $(\widehat{V}_l, \widetilde{V}_l)$ back to the RANDHERD leader.

3. **Challenge (Leader).** The RANDHERD leader aggregates the respective commitments into $\widehat{V} = \prod_{l=0}^{m-1} \widehat{V}_l$ and $\widetilde{V} = \prod_{l=0}^{m-1} \widetilde{V}_l$, and creates two challenges $\widehat{c} = \mathsf{H}(\widehat{V} \parallel w)$ and $\widetilde{c} =$

$\mathsf{H}(\widetilde{V} \parallel \widehat{V})$. Afterwards, the leader sends $(\widehat{c}, \widetilde{c})$ to all group leaders that in turn re-broadcast them to the individual servers of their group.

4. **Response (Servers).** Server i stores the round group challenge $\widehat{c}$ for later usage, creates its individual response $r_i = v_i - \widetilde{c}x_i$, and sends it back to the group leader. The latter aggregates all responses into $\widetilde{r}_l = \sum_{i \in T_l} r_i$ and creates an exception list $\widetilde{E}_l$ of servers in his group that did not respond or sent bad responses. Finally, each group leader sends $(\widetilde{r}_l, \widetilde{E}_l)$ to the RANDHERD leader.

5. **Secret Recovery Request (Leader).** The RANDHERD leader gathers all exceptions $\widetilde{E}_l$ into a list $\widetilde{E}$, and aggregates the responses into $\widetilde{r} = \sum_{l=0}^{m-1} \widetilde{r}_l$ taking $\widetilde{E}$ into account. If at least $2f+1$ servers contributed to $\widetilde{r}$, the RANDHERD leader sends the global group commitment $\widehat{V}$ and the signature $(\widetilde{c}, \widetilde{r}, \widetilde{E})$ to all group leaders thereby requesting the recovery of the group secrets.

6. **Group Secret Recovery (Groups / Servers).** The group leaders re-broadcast the received message. Each group member individually checks that $(\widetilde{c}, \widetilde{r}, \widetilde{E})$ is a valid signature on $\widehat{V}$ and only if it is the case and at least $2f+1$ individual servers signed off, they start reconstructing the short-term secret $\widehat{v}_l$. The group leader creates the group response $\widehat{r}_l = \widehat{v}_l - \widehat{c}\widehat{x}_l$ and sends it to the RANDHERD leader.

7. **Randomness Recovery (Leader).** The RANDHERD leader aggregates all responses $\widehat{r} = \sum_{l=0}^{m-1} \widehat{r}_l$ and, only if he received a reply from all groups, he releases $(\widehat{c}, \widehat{r})$ as the collective randomness of RANDHERD.

Randomness Verification

The collective randomness $(\widehat{c}, \widehat{r})$ of RANDHERD is a collective Schnorr signature on the timestamp w, which is efficiently verifiable against the aggregate group key $\widehat{X}$.

4.4.3 Security Properties

RANDHERD provides the following security properties:

1. **Availability.** Given an honest leader, the protocol successfully completes and produces the final random output Z with high probability.

2. **Unpredictability.** No party learns anything about the final random output Z, except with negligible probability, until the group responses are revealed.

3. **Unbiasability.** The final random output Z represents an unbiased, uniformly random value, except with negligible probability.

4. **Verifiability.** The collective randomness Z is third-party verifiable as a collective Schnorr signature under $\widehat{X}$.

We make the same assumptions as in the case of RANDHOUND (Section 4.3.3) on the behavior of the honest nodes and the cryptographic primitives RANDHERD employs.

RANDHERD uses a simple and predictable ahead-of-time election mechanism to choose the temporary RANDHERD leader in the setup phase. This approach is sufficient because the group assignments and the RANDHERD leader for the randomness phase of the protocol are chosen based on the output of RANDHOUND. RANDHOUND's properties of unbiasability and unpredictability hold for honest and dishonest clients. Therefore, the resulting group setup has the same properties in both cases.

Availability. Our goal is to ensure that with high probability the protocol successfully completes, even in the presence of an active adversary.

As discussed above, the use of RANDHOUND in the setup phase ensures that all groups are randomly assigned. If the RANDHERD leader makes satisfactory progress, the secret sharing threshold $t_l = f/m + 1$ enables $2f/m$ honest nodes in each group to reconstruct the short-term secret $\widehat{v}_l$, hence produce the group response $\widehat{r}_l$ without requiring the collaboration of malicious nodes. An honest leader will make satisfactory progress and eventually, output $\widehat{r}$ at the end of step 7. This setup corresponds to a run of RANDHOUND by an honest client. Therefore, the analysis of the failure probability of a RANDHOUND run described in Section 4.5.3 is applicable to RANDHERD in the honest leader scenario.

In RANDHERD, however, with a probability f/n, a dishonest client will be selected as the RANDHERD leader. Although the choice of a dishonest leader does not affect the group assignments, he might arbitrarily decide to stop making progress at any point of the protocol. We need to ensure RANDHERD's availability over time, and if the current leader stops making adequate progress, we move to the next leader indicated by the random output of RANDHOUND and, as with common BFT protocols, we rely on *view change* [55, 149] to continue operations.

Unpredictability. We want to ensure that the random output of RANDHERD remains unknown until the group responses $\widehat{r}_l$ are revealed in step 6.

The high-level design of RANDHERD closely resembles that of RANDHOUND. Both protocols use the same thresholds, assign n nodes into m groups, and each group contributes exactly one secret towards the final random output of the protocol. Therefore, as in RANDHOUND, there will similarly be at least one RANDHERD group with at least an average number of honest nodes. Furthermore, the secret-sharing and required group inputs threshold of $t_l = f + 1$ guarantees that for at least one group, the adversary cannot prematurely recover $\widehat{v}_l$ and reconstruct the group's response $\widehat{r}_l$. Therefore, before step 6, the adversary will control at most $m - 1$ groups and obtain at most $m - 1$ out of m responses that contribute to $\widehat{r}$.

Unbiasability. Our goal is to prevent the adversary from biasing the value of the random output $\widehat{r}$.

As in RANDHOUND, we know that for at least one group the adversary cannot prematurely

recover $\widehat{r}_l$ and that $\widehat{r}_l$ contains a contribution from at least one honest group member. Further, the requirement that the leader must obtain a sign-off from $2f+1$ individual nodes in step 4 on his commitment $\widehat{V}$, fixes the output value $\widehat{r}$ before any group secrets $\widehat{r}_l$ are produced. This effectively commits the leader to a single output $\widehat{r}$.

The main difference between RANDHOUND and RANDHERD is the fact that an adversary who controls the leader can affect unbiasability by withholding the protocol output $\widehat{r}$ in step 7, if $\widehat{r}$ is not beneficial to him. A failure of a leader would force a view change and therefore a new run of RANDHERD, giving the adversary at least one alternative value of $\widehat{r}$, if the next selected leader is honest, or several tries if multiple successive leaders are dishonest or the adversary can successfully DoS them. The adversary cannot freely choose the next value of $\widehat{r}$, nor go back to the previous value if the next one is not preferable, the fact that he can sacrifice a leadership role to try for an alternate outcome constitutes bias. This bias is limited, as the view-change schedule must eventually appoint an honest leader, at which point the adversary has no further bias opportunity. Section 4.4.4 further addresses this issue with an improvement ensuring that an adversary can hope to hold leadership for at most $O(\log n)$ such events before permanently losing leadership and hence bias opportunity.

Verifiability. The random output $\widehat{r}$ generated in RANDHERD is obtained from a TSS-CoSi Schnorr signature $(\widehat{c}, \widehat{r})$ on input w against a public key $\widehat{X}$. Any third-party can verify $\widehat{r}$ by simply checking the validity of $(\widehat{c}, \widehat{r})$ as a standard Schnorr signature on input w using $\widehat{X}$.

4.4.4 Addressing Leader Availability Issues

Each run of RANDHERD is coordinated by a RANDHERD leader who is responsible for ensuring satisfactory progress of the protocol. Although a (honest or dishonest) leader might fail and cause the protocol failure, we are specifically concerned with intentional failures that benefit the adversary and enable him to affect the protocol's output.

As discussed above, once a dishonest RANDHERD leader receives responses from group leaders in step 7, he is the first one to know $\widehat{r}$ and can act accordingly, including failing the protocol. However, the failure of the RANDHERD leader does not necessarily have to cause the failure of the protocol. Even without the dishonest leader's participation, $f/m+1$ of honest nodes in each group are capable of recovering the protocol output. They need, however, a consistent view of the protocol and the output value that was committed to.

Instead of requiring a CoSi round to get $2f+1$ signatures on $\widehat{V}$, we use a Byzantine Fault Tolerance (BFT) protocol to reach consensus on $\widehat{V}$ and consequently on the global challenge $\widehat{c} = \mathsf{H}(\widehat{V} \parallel w)$. Upon a successful completion of BFT, at least $f+1$ honest nodes have witnessed that we have consensus on the $\widehat{V}$. Consequently, the $\widehat{c}$ that is required to produce each group's response $\widehat{r}_l = \widehat{v}_l - \widehat{c}\widehat{x}_l$ is "set in stone" at this point. If a leader fails, instead of restarting RANDHERD, we can select a new leader, whose only allowed action is to continue the protocol from the existing commitment. This design removes the opportunity for a dishonest leader

biasing the output even a few times before losing leadership.

Using a traditional BFT protocol (*e.g.,* PBFT [55]) would yield poor scalability for RANDHERD because of the large number of servers that participate in the protocol. To overcome this challenge, we use BFT-CoSi from ByzCoin [149], a Byzantine consensus protocol that uses scalable collective signing, to agree on successfully delivering the commitment $\widehat{V}$. Due to the BFT guarantees RANDHERD crosses the point-of-no-return when consensus is reached. Even if the dishonest leader, tries to bias output by failing the protocol, the new (eventually honest) leader will be able to recover $\widehat{r}$, allowing all honest servers to successfully complete the protocol.

The downside of this BFT-commitment approach is that once consensus is reached and the point-of-no-return is crossed, then in the rare event that an adversary controls two-thirds of any group, the attacker can halt the protocol forever by preventing honest nodes from recovering the committed secret. This risk may necessitate a more conservative choice of group size, such that the chance of an adversary ever controlling any group is not merely unlikely but truly negligible.

4.4.5 Extensions

Randomizing Temporary-Leader Election

The current set-up phase of RANDHERD uses a simple leader election mechanism. Because the ticket generation uses only values known to all nodes, it is efficient as it does not require any communication between the nodes but makes the outcome of the election predicable as soon as the cothority configuration file C is available. We use this mechanism to elect a temporary RANDHERD leader whose only responsibility is to run and provide the output of RANDHOUND to other servers. RANDHOUND's unbiasability property prevents the dishonest leader from biasing its output. However, an adversary can force f restarts of RANDHOUND and can, therefore, delay the setup by compromising the first (or next) f successive leaders in a well-known schedule.

To address this issue, we can use a lottery mechanism that depends on verifiable random functions (VRFs) [180], which ensures that each participant obtains an unpredictable "fair-share" chance of getting to be the leader in each round. Each node produces its lottery ticket as $t_i = \mathsf{H}(C \parallel j)^{x_i}$, where C is the group configuration, j is a round number, and x_i is node i's secret key, along with a NIZK consistency proof showing that t_i is well-formed. Since an adversary has at least a constant and unpredictable chance of losing the leadership to some honest node in each lottery, this refinement ensures with high probability that an adversary can induce at most $O(\log n)$ successive view changes before losing leadership.

BLS Signatures

Through the use of CoSi and TSS, RANDHERD utilizes collective Schnorr signatures in a threshold setting. Other alternatives are possible. Specifically, Boneh-Lynn-Shacham (BLS) [39] signatures require pairing-based curves, but offer even shorter signatures (a single elliptic curve point) and a simpler signing protocol. In the simplified design using BLS signatures, there is no need to form a fresh Schnorr commitment collectively, and the process does not need to be coordinated by a group leader. Instead, a member of each subgroup, whenever it has decided that the next round has arrived, produces and releases its share for a BLS signature of the message for the appropriate time (based on a hash of view information and the wall-clock time or sequence number). Each member of a given subgroup waits until a threshold number of BLS signature shares are available for that subgroup, and then forms the BLS signature for this subgroup. The first member to do so can then simply announce or gossip it with members of other subgroups, combining subgroup signatures until a global BLS signature is available (based on a simple combination of the signatures of all subgroups). This activity can be unstructured and leaderless, since no "arbitrary choices" need to be made per-transaction: the output of each time-step is completely deterministic but cryptographically random and unpredictable before the designated time.

4.5 Evaluation

This section experimentally evaluates our prototype implementations of RANDHOUND and RANDHERD. The primary questions we wish to evaluate are whether architectures of the two protocols are practical and scalable to large numbers, *e.g.*, hundreds or thousands of servers, in realistic scenarios. Important secondary questions are what the important costs are, such as randomness generation latencies and computation costs. We start with some details on the implementation itself, followed by our experimental results, and finally, describe our analysis of the failure probability for both protocols.

4.5.1 Implementation

We implemented PVSS, TSS, RANDHOUND, and RANDHERD in Go [118] and made these implementations available on GitHub as part of the EPFL DEDIS lab's Cothority framework.[2] We reused existing cothority framework code for CoSi and network communication, and built on the DEDIS advanced crypto library[3] for cryptographic operations such as Shamir secret sharing, zero-knowledge proofs, and optimized arithmetic on the popular Curve25519 elliptic curve [25]. As a rough indicator of implementation complexity, Table 4.1 shows approximate lines of code (LoC) of the new modules. Line counts were measured with GoLoC.[4]

[2] https://github.com/dedis/cothority
[3] https://github.com/dedis/crypto
[4] https://github.com/gengo/goloc

Table 4.1 – Lines of code per module

PVSS	TSS	RANDHOUND	RANDHERD
300	700	1300	1000

4.5.2 Performance Measurements

Experimental Setup

We ran all our experiments on DeterLab[5] using 32 physical machines, each equipped with an Intel Xeon E5-2650 v4 (24 cores at 2.2 GHz), 64 GBytes of RAM, and a 10 Gbps network link. To simulate a globally-distributed deployment realistically, we restricted the bandwidth of all intern-node connections to 100 Mbps and imposed 200 ms round-trip latencies on all communication links.

To scale our experiments up to 1024 participants given limited physical resources, we oversubscribed the DeterLab servers by up to a factor of 32, arranging the nodes such that most messages had to go through the network. To test the influence of oversubscription on our experiments, we reran the same simulations with 16 servers only. This resulted in an overhead increase of about 20%, indicating that our experiments are already CPU-bound and not network-bound at this scale. We, therefore, consider these simulation results to be pessimistic: real-world deployments on servers that are not oversubscribed in this way may yield better performance.

RANDHOUND

Figure 4.4 shows the CPU-usage costs of a complete RANDHOUND run that generates a random value from N servers. We measured the total costs across all servers, plus the costs of the client that coordinates RANDHOUND and generates the Transcript. With 1024 nodes divided into groups of 32 nodes, for example, the complete RANDHOUND run to generate randomness requires less than 10 CPU minutes total, corresponding to a cost of about $0.02 on Amazon EC2. This cost breaks down to about 0.3 CPU seconds per server, representing negligible per-transaction costs to the servers. The client that initiates RANDHOUND spends about 3 CPU minutes, costing less than $0.01 on Amazon EC2. These results suggest that RANDHOUND is quite economical on today's hardware.

Figure 4.5 shows the wall clock time of a complete RANDHOUND run for different configurations. This test measures total time elapsed from when the client initiates RANDHOUND until the client has computed and verified the random output. Our measurements show that the wall clock time used by the servers to process client messages is negligible in comparison, and hence not depicted in Figure 4.5. In the 1024-node configuration with groups of 32 nodes,

[5] http://isi.deterlab.net/

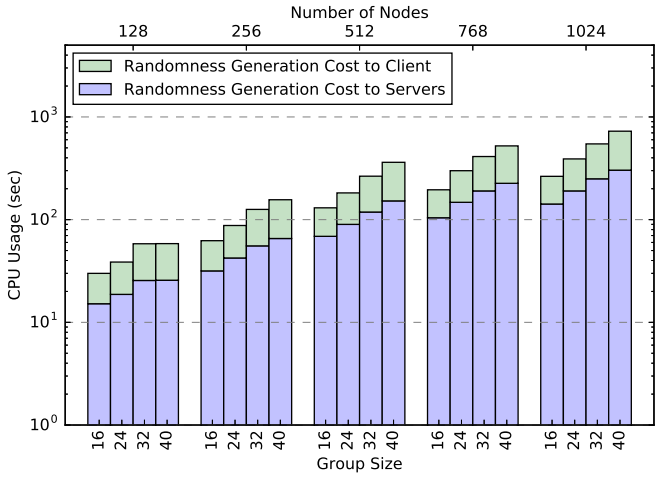

Figure 4.4 – Overall CPU cost of a RANDHOUND protocol run

randomness generation and verification take roughly 290 and 160 seconds, respectively.

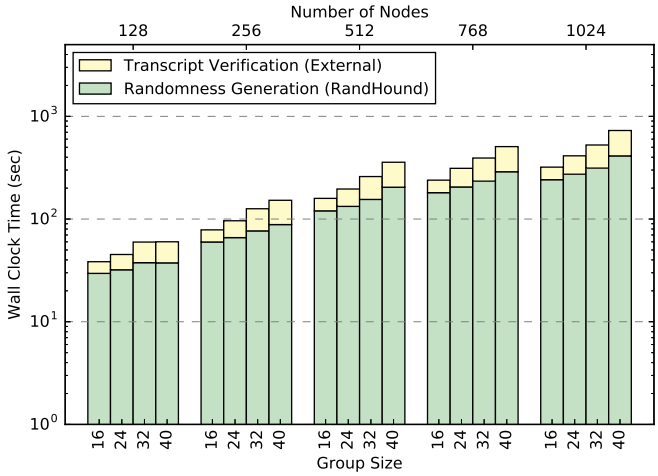

Figure 4.5 – Total wall clock time of a RANDHOUND protocol run

RANDHERD

The RANDHERD protocol requires a setup phase, which uses RANDHOUND to form random groups and CoSi to sign the RANDHERD collective key. The measured CPU usage of RANDHERD setup is depicted in Figure 4.6. For 1024 nodes and a group size of 32, RANDHERD setup requires roughly 40 CPU-hours total (2.3 CPU-minutes per node), corresponding to a cost of $4.00 total on Amazon EC2 (0.3 cents per participant). The associated wall clock time we measured, not depicted in the graphs, amounts to about 10 minutes.

After this setup, RANDHERD produces random numbers much more efficiently. Figure 4.7

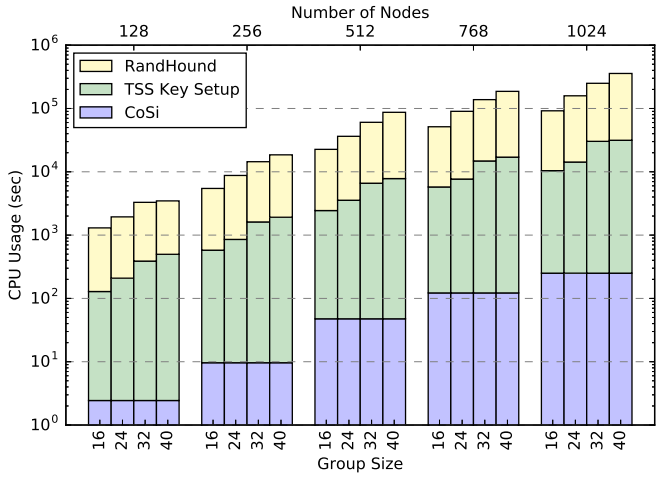

Figure 4.6 – Total CPU usage of RANDHERD setup

illustrates measured wall clock time for a single RANDHERD round to generate a 32-byte random value. With 1024 nodes in groups of 32, RANDHERD takes about 6 seconds per round. The corresponding CPU usage across the entire system, not shown in the graphs, amounts to roughly 30 seconds total (or about 29 CPU-milliseconds per node).

A clear sign of the server-oversubscription with regard to the network-traffic can be seen in Figure 4.7, where the wall clock time for 1024 nodes and a group size of 32 is lower than the one for a group size of 24. This is due to the fact that nodes running on the same server do not have any network-delay. We did a verification run without server oversubscription for up to 512 nodes and could verify that the wall clock time increases with higher group-size.

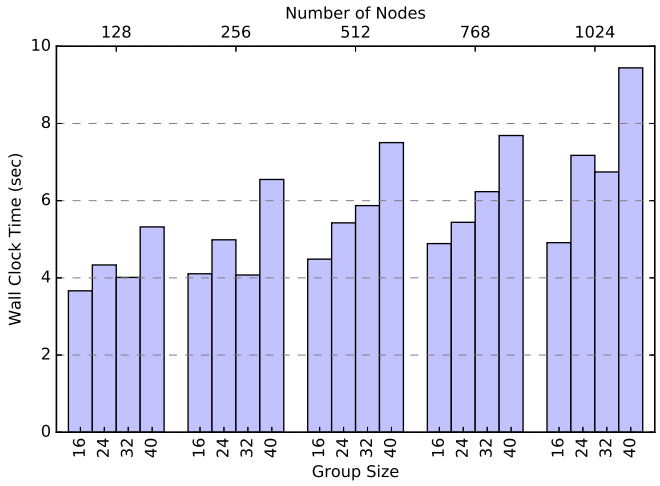

Figure 4.7 – Wall clock time per randomness creation round in RANDHERD

Figure 4.8 compares communication bandwidth costs for CoSi, RANDHOUND, and RANDHERD, with varying number of participants and a fixed group size of 32 nodes. The straight lines

depict total costs, while the dashed lines depict the average cost per participating server. For the case of 1024 nodes, CoSi and RandHound require about 15 and 25 MB, respectively. After the initial setup, one round of RandHerd among 1024 nodes requires about 400 MB (excluding any setup costs) due to the higher in-group communication. These values correspond to the sum of the communication costs of the entire system and, considering the number of servers involved, are still fairly moderate. This can be also seen as the average per server cost is less than 300 KB for RandHerd and around 20 KB for CoSi and RandHound.

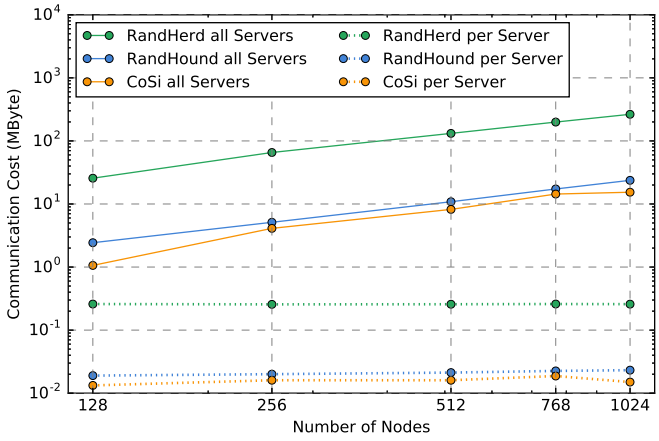

Figure 4.8 – Comparison of communication bandwidth costs between RandHerd, RandHound, and CoSi for fixed group size $c = 32$

Finally, Figure 4.9 compares RandHerd, configured to use only one group, against a non-scalable baseline protocol similar to RandShare. Because RandShare performs verifiable secret sharing among all n nodes, it has computation and communication complexity of $O(n^3)$. In comparison, RandHerd has sublinear per-round complexity of $O(\log n)$ when the group size is constant.

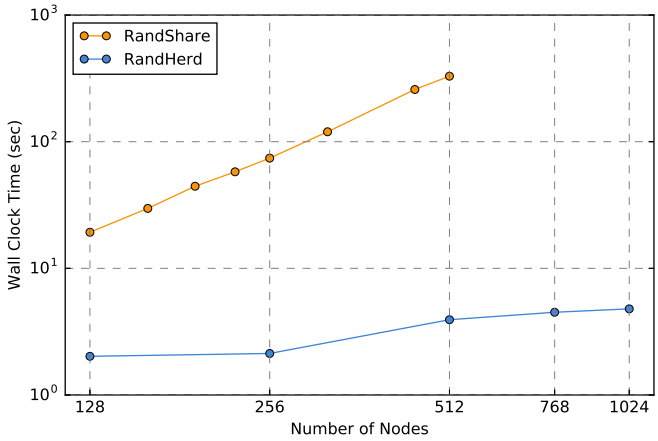

Figure 4.9 – Comparison of randomness generation times for RandShare and RandHerd (group size $c = 32$ for RandHerd and $c = n$ for RandShare)

4.5.3 Availability Failure Analysis

An adversary who controls too many nodes in any group can compromise the availability of both RANDHOUND and RANDHERD. We can analyze the probability of availability failure assuming that nodes are assigned randomly to groups, which is the case in RANDHOUND when the client assigns groups honestly, and is always the case in RANDHERD. As discussed in Figure 4.3.3, dishonest grouping in RANDHOUND amounts to self-DoS by the client and is thus outside the threat model.

To get an upper bound for the failure probability of the entire system, we first bound the failure probability of a single group, that can be modeled as a random variable X that follows the hypergeometric distribution, followed by the application of Boole's inequality, also known as the union bound. For a single group we start with Chvátal's formula [232]

$$P[X \geq E[X] + cd] \leq e^{-2cd^2}$$

where $d \geq 0$ is a constant and c is the number of draws or in our case the group size. The event of having a disproportionate number of malicious nodes in a given group is modeled by $X \geq c - t + 1$, where t is the secret sharing threshold. In our case we use $t = cp + 1$ since $E[X] = cp$, where $p \leq 0.33$ is the adversary's power. Plugging everything into Chvátal's formula and doing some simplifications, we obtain

$$P[X \geq c(1-p)] \leq e^{-2c(1-2p)^2}$$

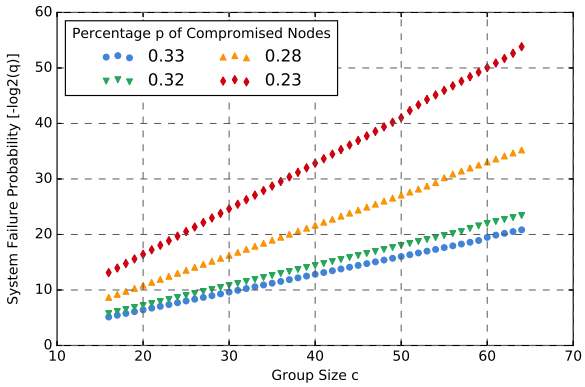

Figure 4.10 – System failure probability for varying group sizes

Applying the union bound on this result, we obtain Figure 4.10 and and Figure 4.11, which show average system failure probabilities q for varying group sizes ($c = 16, \ldots, 64$) and varying adversarial power ($p = 0.01, \ldots, 0.33$), respectively. Note that q on the y-axis is plotted in "security parameter" form as $-\log_2(q)$: thus, higher points in the graph indicate exponentially lower failure probability. Finally, Figure 4.2 lists failure probabilities for some concrete configurations. There we see, for example, that both RANDHOUND and RANDHERD have a failure

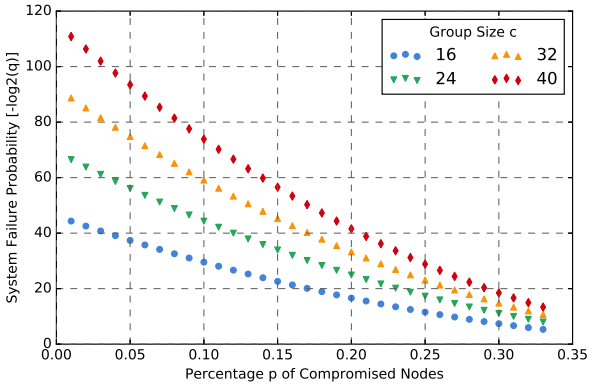

Figure 4.11 – System failure probability for varying adversarial power

probability of at most $2^{-10.25} \approx 0.08\%$ for $p = 0.33$ and $c = 32$. Moreover, assuming $p = 0.33$, we identified the point where the system's failure probability falls below 1% for a group size of $c = 21$.

Table 4.2 – System failure probabilities q (given as $-\log_2(q)$) for concrete configurations of adversarial power p and group size c

$p \mid c$	16	24	32	40
0.23	13.13	19.69	26.26	32.82
0.28	8.66	15.17	17.33	21.67
0.33	5.12	7.69	10.25	12.82

4.6 Conclusions

Although many distributed protocols critically depend on public bias-resistant randomness for security, current solutions that are secure against active adversaries only work for small ($n \approx 10$) numbers of participants [49, 141]. In this work, we have focused on the important issue of scalability and addressed this challenge by adapting well-known cryptographic primitives. We have proposed two different approaches to generating public randomness in a secure manner in the presence of a Byzantine adversary. RandHound uses PVSS and depends on the pigeonhole principle for output integrity. RandHerd relies on RandHound for secure setup and then uses TSS and CoSi to produce random output as a Schnorr signature verifiable under a collective RandHerd key. RandHound and RandHerd provide *unbiasability*, *unpredictability*, *availability* and *third-party verifiability* while retaining good performance and low failure probabilities. Our working prototype demonstrates that both protocols, in principle, can scale even to thousands of participants. By carefully choosing protocols parameters, however, we achieve a balance of performance, security, and availability. While retaining a failure probability of at most 0.08% against a Byzantine adversary, a set of 512 nodes divided into groups of 32 can produce fresh random output every 240 seconds in RandHound, and every 6 seconds in RandHerd after initial setup.

5 Decentralized Tracking and Long-Term Relationships using SKIPPER

5.1 Introduction

Today's network services and applications commonly maintain long-term relationships with users and their devices through centralized account databases and login credentials (*e.g.,* username/password). Users' devices, in turn, rely on many other centralized services each time they access such a network application: *e.g.,* using centralized DNS [188] or DNSSEC [87] servers to look up the server's IP address, using digital certificates signed by central Certificate Authorities (CA) [66] to authenticate the server, and using centralized storage such as iCloud keychains [238] to cache login credentials to the user's network applications. Centralized network applications and their user databases have become prime targets enabling hackers to compromise the security and privacy of millions of users at once [20, 244]. Further, centralized authorities required to connect to these applications, such as Certificate Authorities, have become a regular target for nation-state attackers [65]. Users cannot even rely on the software they run locally, because they use centralized software update services "secured" with code-signing keys that are readily available to malware authors on the black market [143] – and have even been accidentally published [186].

Less-common decentralized application architectures, such as peer-to-peer file-sharing [61, 24] and cryptocurrencies [191, 250], enable users to participate anonymously or pseudonymously without relying on centralized authorities or accounts. However, a key technical challenge that has limited the security and privacy benefits of decentralized architectures in practice is the problem of *maintaining long-term relationships* between a decentralized application and its users' many client devices. Clients must be able to track the decentralized application's state securely over the short-term, *e.g.,* as transactions are committed to a blockchain; over the medium-term, *e.g.,* as the set of available servers or active blockchain miners change dynamically; and over the long-term, *e.g.,* as client- and server-side software and cryptographic keys evolve independently or are replaced due to security incidents.

Existing solutions to this *decentralized relationship* problem impose on clients either the high resource costs of "first-class" participation in a heavyweight decentralized protocol, or

the security cost of "re-centralization" by delegating critical functions to, and hence trusting, centralized servers. As a classic example of this tradeoff, the originally "flat" peer-to-peer topology of Freenet [61] gave way for efficiency reasons to designs in which clients delegated search and other functions to more powerful *supernodes*, which lightened clients' loads but could also potentially censor or poison content forwarded to clients. Similarly, to avoid relying on central authorities entirely, a Bitcoin [191] client must follow the blockchain itself, incurring the high bandwidth cost of downloading blocks and the high CPU costs of verifying them, just to check that a payment transaction has committed. In practice "light" clients simply trust centralized Bitcoin "full nodes" to tell them if a transaction has committed – or even just trust a Bitcoin exchange for complete management of the user's Bitcoin wallet much like a traditional bank.

We address the decentralized relationship challenge with SKIPPER, a blockchain-inspired architecture enabling decentralized or partly-decentralized services to adapt and evolve at diverse time-scales, while ensuring that resource-constrained clients can efficiently and securely track the service's evolution. SKIPPER enables a resource-constrained client device to find and authenticate the service's latest configuration even after being offline for weeks or months and to verify that the set of servers (configuration) he trusts is fresh (not vulnerable to replay attacks).

These capabilities make SKIPPER the first architecture we are aware of enabling a decentralized service to maintain secure long-term relationships efficiently with a large number of clients, without clients needing either to trust central authorities or to participate continually in heavyweight protocols. While designed with highly decentralized systems in mind, SKIPPER's decentralized relationship management may also be useful to centrally-managed services wishing to reduce the need to trust widely-distributed client-facing infrastructure, such as in mirrored software distribution/update systems [75, 255, 132], Web content delivery networks (CDNs) [5], or centrally-banked cryptocurrencies [74].

SKIPPER's key technical contribution is *Skipchains*, a blockchain-like structure enabling clients to follow and efficiently navigate arbitrarily-long service configuration timelines, both forward (*e.g.,* to validate a newer service configuration than the last one the client has seen) and backward (*e.g.,* to verify that a particular transaction was committed at some historical time). Skipchains are naturally inspired by skip lists [207, 190], but efficiently support traversal both backward and forward in time by using hash-links to represent pointers into the past, and using multisignatures [221, 38] to create pointers into the future. By building on the CoSi protocol for scalable collective signing [243], many independent participants implementing a SKIPPER service (*e.g.,* blockchain miners) can separately validate and "sign off" on each forward link, contributing to a compressed multisignature that is almost as compact and cheap to verify as a single conventional digital signature. Because these collective signatures are offline-verifiable, resource-constrained clients need not track a blockchain continuously like a Bitcoin full node, but can privately exchange, gossip, and independently validate arbitrary newer or older blocks or transactions on-demand. While multisignatures and collective

signing protocols are pre-existing techniques, this chapter's contribution is their use to create the efficiently navigable Skipchain data-structure.

Second, skipchains can represent not only application timelines that users track, but also user-controlled *identity timelines* that applications track in order to authenticate users and maintain per-user state, thus providing a more decentralized and user-controlled alternative to centrally-managed and frequently-hacked login/password databases. Finally, SKIPPER explores the use of skipchains in multiple roles, *e.g.,* using a *root-of-trust* skipchain secured using offline keys for long-term service evolution or to validate more rapidly-changing configuration and/or transaction skipchains secured using online keys.

Our prototype implementation of SKIPPER demonstrates the use and benefits of skipchains in a simple proof-of-concept application of a software update service. For example, skipchains can increase the security of PyPI updates with minimal overhead, whereas a strawman approach would incur the increase of 500%. Further, our prototype uses skipchains to represent distributed user-identities as sets of per-device SSH [252] key-pairs, enabling users to manage per-device keys more easily and enabling applications to track their users' key-configuration changes securely and seamlessly without relying on username/password databases.

In summary, the key contributions SKIPPER makes are (a) skipchains, a novel timeline-tracking structure enabling users to track applications and vice versa without trusting centralized intermediaries, (b) use of skipchains "in reverse" to allow applications to authenticate users via distributed user identities.

5.2 Motivation

In this Section, we summarize the systems that motivated the development of SKIPPER.

5.2.1 The Relationship Problem

A fundamental problem of distributed architectures may be summarized as *relationship-keeping*: enabling client devices, representing users, to connect securely with servers elsewhere in the network that can satisfy the clients' requests, and similarly allowing applications to authenticate their users and maintain per-user state. Relationship-keeping in today's hostile Internet is challenging because (a) attackers in the network might control some of the endpoints and be able to compromise these interactions, *e.g.,* by impersonating clients to servers and vice versa; (b) client software (whether web browsers or downloaded applications) is often only loosely synchronized with evolving server software and their security configurations (*e.g.,* certificates); and (c) services must scale gracefully to handle the client-imposed load, balancing requests across a dynamically varying set of available servers, without compromising security.

Today's widely-deployed infrastructure addresses this relationship challenge by relying perva-

sively on indirection through centralized authorities, thereby creating many single points of failure and high-value attack targets.

> "All problems in computer science can be solved by another level of indirection"
>
> — David Wheeler

Clients download software and updates signed by software vendors and repositories, which may be compromised [52], coerced [195, 103], or lose their keys [186]. This software then uses centralized directories [188, 87] to find and connect with a service, and uses centralized certificate authorities to secure these connections [81], but both directory and certificate authorities are prone to security failures [44, 45]. Finally, typical cloud-based services just push the problem deeper without solving it, by excessively relying on the "indirection principle" to scale. They pass connections through centralized load-balancing front-ends [189] and/or delegate authentication mechanisms to centralized servers within the provider's infrastructure [136], each of which represents single points of failure and prime attack targets [19].

Decentralized architectures attempt to avoid such centralization and associated single points of failure, but have not yet proven sufficiently general, secure, or scalable in practice. Certificate Transparency (CT) [163] proposes a solution of forcing certificates to be publicly logged and audited. This approach, however, can only offer weak security in the absence of network partition and eclipse attacks as it can only retroactively detect misbehavior. Unstructured peer-to-peer systems [61, 131] and distributed hash tables [236, 216] attempted to offer decentralized search and connectivity protocols, but became widely deployed only in centrally-managed contexts [161, 76] due to persistent weaknesses in decentralized operation [231] such as Sybil attacks [84] and content poisoning attacks [169]

Bitcoin [191] reinvigorated interest in decentralized architectures, mitigating the Sybil attack problem via competitive proof-of-work "mining." While groundbreaking and inspiring exciting new directions such as smart contracts [250, 153], decentralized blockchain architectures face major efficiency and scalability challenges [70], and are currently nowhere near efficient or scalable enough to replace current centralized network service architectures. Further, the incentive structure for mining has caused Bitcoin and other currencies to re-centralize in practice, until only two or three miners (sometimes just one) could effectively control the blockchain. Finally, current blockchain architectures fail to extend security guarantees efficiently to resource-constrained "light" clients, which must either suffer inordinate bandwidth and resource costs to track the blockchain directly, or sacrifice decentralization by placing trust in centralized full nodes or exchange services.

5.2.2 Motivating Examples

In this section, we mention some systems that can increase their security and usability with SKIPPER.

Software Updates The automatic detection and installation of software updates is one of the most common tasks of every computer. There are multiple agents implementing this functionality, like package managers [75, 255, 132], and library managers [209], however these are proven to be insecure [158]. This happens due to the infrequent key rotation, that leads to long attack windows. Furthermore, the lack of transparency in the whole process and the existence of few centrally controlled signing keys, allows the existence of forgery request from state level adversaries [103].

To show that it is possible to use SKIPPER with low overhead we want to focus on a particular system and increases its security. TUF [217, 158] is built to secure software updates, by using delegations and is considered the state-of-the-art. TUF decouples the multiple roles that a software repository has to serve into multiple trust domains, each of which has a specific key or set of keys. This way there can be different trade-offs between security and usability. Keys that need to be used often, are kept online but their compromise can do little damage. On the other hand, security-critical keys are kept offline and rarely used because the attacker could fully impersonate the software repository upon compromise.

Both online and offline keys need to rotate periodically in order to remain secure, but there are many clients that update infrequently. These clients need a way to get securely up-to-date in a trustless environment assuming that an attacker has compromised some of the keys they used to trust. Additionally, there is no anti-equivocation mechanism, which means that a dishonest or coerced repository can create multiple versions of the same software, for specific clients without being detected. Finally, in TUF the updates are not linked together which means that a victim cannot detect that he was attacked even after recovery, which allows an attacker to install malware in the victim's system and then activate it whenever he wants.

User Identities One of the problems that emerged by the spurt of the Internet of Things and cloud computing, is that every user owns multiple identities with which she wants to access multiple remote locations. These accounts are not static or eternally secure, meaning that the user needs to take care of correct access control and manage all her digital identities.

As a motivation to this work, we look into handling user identities and specifically tracking the development of a user's accounts and connected public keys. The goal is to enable tracking of changes in the identity of the user like email addresses and public keys. This is applicable in multiple systems like PGP [90] or SSH [252], but we specifically focused on SSH.

SSH has a trade-off between usability and security, with usability being almost always preferred. The average user creates SSH keys at her laptop (frequently without a pass-phrase) and copies the private key to every other device she uses. Then she uploads the public key in the `authorized_keys` configuration file of servers she wants to remotely access, granting root privileges to it. Although this is easy since the user copies the private key to her new laptop and everything works, it doesn't do much for security. If an attacker steals a device, he has physical access to the SSH key. Furthermore, since a change of the SSH key would impose

a big overhead, the first SSH key is eternal, providing the attacker with eternal, unbounded access. This unbounded root access means that the attacker has write privileges to the `authorized_keys`, giving him the ability to lock out the legitimate user.

On the other hand, the recommended practice [192] is to create one key per host and limit the damage an attacker can do when compromising one of the user's devices, by configuring the key's privileges to be a subset of the root privileges. If this is combined with the frequent rotation of keys, it can already guarantee a higher level of security. However, this practice requires a user to update the access lists of numerous servers. Finally, if a highly trusted key (*e.g.,* laptop's) is compromised, the attacker can still get root access and update the `authorized_keys` to exclude the actual user.

5.3 Overview

In this section, we introduce SKIPPER, a framework for secure and efficient decentralized relationship-keeping. First, we give an overview of the architecture and then introduce the notion of timelines and tracking.

5.3.1 Security Goals and Threat Model

The goal of SKIPPER is to enable secure tracking of an evolving server-side timeline, in a decentralized environment, without the need to trust centralized intermediaries such as lookup services or certificate authorities. The server side can change, partially or completely, its representative public keys, in arbitrary timings without jeopardizing the security of the clients. After bootstrapping, the client should be able to autonomously retrieve and verify the full timeline (or the useful parts), without needing to trust a "special supernode". Furthermore, a client that has stopped tracking the timeline updates should be able to securely, autonomously and efficiently retrieve the last version of the system. Finally, SKIPPER enables a client to be certain before accepting any update that many independent observers have witnessed and logged the update and have verified that there is no equivocation (no other valid timeline) If an attempt to fork the timeline is detected, this will signify a compromised key and SKIPPER should provide means to defend or recover quickly, whistleblowing the problem to any affected clients and protecting everybody else.

The threat model of the system considers an adversary that is able to compromise only online keys and can respond to client requests. Furthermore, any given group of "trustees" has a maximum number of $\frac{1}{3}$ of Byzantine Faults.

5.3.2 Architectural Model and Roles

Figure 5.1 illustrates SKIPPER's abstract architectural model, which is conceptually independent of particular applications. We explore later how SKIPPER maps to particular example

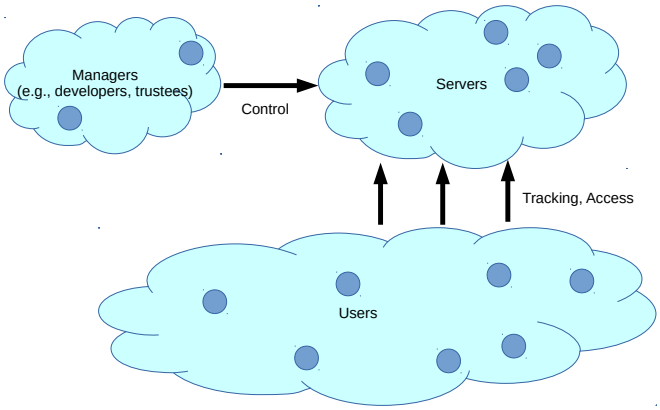

Figure 5.1 – Architectural Roles in SKIPPER

application domains.

Participants in a SKIPPER service play three logical roles, which we term *client*, *server*, and *manager*. Clients represent users of the service, who need to track the service's state and maintain a relationship with the servers representing it. The servers actually implement the critical functions of the service, producing a *timeline* that the clients follow to track configuration changes and submit requests. The managers represent whatever authority has (perhaps collective) power to decide and authorize changes to the service's configuration, such as by adding or removing participants, revoking or updating cryptographic keys, and rolling out new software or protocol versions either incrementally or *en masse*. Some SKIPPER participants may play multiple roles: *e.g.*, a server may also be a client, and a client may also be a manager.

This simple three-role model is compatible with many types of decentralized systems. For example, in Bitcoin [191] and related cryptocurrencies [250, 74, 149], the clients are user devices wishing to send and receive payments, the servers are the miners who extend the blockchain and enforce its consistency, and the managers are the developers who decide what protocol and software changes to roll out at which times. In the Tor anonymity system [82], the clients are users that request and use anonymous communication circuits, the servers are the public Tor relays that accept these requests and forward traffic over these circuits, and the managers are the ten semi-trusted *directory authorities* who control the list of relays and the bandwidth-capacity information that clients use to pick relays when creating circuits.

We also intend SKIPPER to be usable by – and potentially useful to – services that are centrally managed, but wish to have a decentralized client-facing "service cloud" for security, privacy, scalability, or other reasons. In this case, we still apply the model in Figure 5.1, but with only one entity (or its representatives) serving the manager role. One example is software update services, in which a single company or tight-knit group of developers serves as the manager who creates a timeline of versions of a software package or repository, which must

be distributed to and validated by a large number of clients through a network of preferably untrusted mirror sites. Web CDNs [5, 63] offer a related example, where one (centralized) owner of a website desires the CDN's help in offloading content distribution load and attack protection. While current CDNs themselves are centralized and must be trusted by the website owners using them, some website owners might prefer to decentralize the client-facing CDN role and reduce the trust that must be placed in any one entity [181, 155].

5.3.3 Timelines and Tracking

We refer to *timeline* as a list of updates to the state of a system. These updates can be the evolution of the roster of agents ensuring the correctness of the application (configurations), or updates of the application state (data) as the service operates.

Tracking is the action of following a timeline. Both servers and clients track configuration and data timelines for different purposes. Servers are defined by the configuration timeline and rather create it than track it, so that the system remains correct. A subset of servers is also the group that allows for the append of new updates at the data timeline securely, and the full set of servers opportunistically track the data timeline to verify the correctness. Contrary, clients track the data timeline to retrieve the latest application state and track the configuration timeline to verify that the data they retrieved are verified by the correct agents.

For tracking to be secure, timelines need to be created, so that a client can crawl and verify the timeline without needing to trust centralized authorities or other intermediaries. Given a secure reference to (*e.g.,* a cryptographic hash of) anyone correct point in the timeline, the client should be able to find the latest correct point securely – *e.g.,* the latest official version of a software package or CDN-mirrored website – without having to trust any intermediary. Further, given a secure reference to any point in the timeline, the client should efficiently be able to verify the existence of any other point in the timeline proposed by another client: *e.g.,* to verify a committed transaction representing a payment in a cryptocurrency blockchain.

5.4 Design of SKIPPER

In this section, we introduce the concept of *skipchains*, the core components of SKIPPER. We start from a base case that is composed of a tamper-evident log managed by a central authority. Afterwards, we transform this construction into skipchains step-by-step by addressing the following challenges:

1. **Key Evolution** We add forward linking to enable key rotation and trust delegation [137] in a secure manner.

2. **Decentralization** We introduce decentralization via cothorities [243] which allows to audit the actions of cothority members and eliminates forking.

3. **Efficiency** We utilize techniques from skiplists [207, 190] to be able to skip multiple steps in either direction of the chain enabling logarithmic instead of linear timeline traversal.

5.4.1 Centrally Managed Tamper-Evident Logs

The simplest way to implement a timeline is in the form of a log. Every new update is logged and anyone tracking the head of the log becomes immediately aware of any changes. By using tamper-evident logs [71, 163, 168] one can provide some protection in case of a key compromise of the log-managing authority. We implement our tamper-evident log as a blockchain [191] where every update is put into a block together with a back-link in the form of a cryptographic hash of the previous block. As a result, each new block depends on every previous one and there is no way to tamper with past blocks. A client which wants to catch up with the latest updates can request missing blocks from the authority and then crawl through the back-links until it finds the last known block. This way the client can verify that the authority provided a linear history.

5.4.2 Anti-Equivocation via Collective Witnessing

One of the problems of trusting a centralized authority to maintain SKIPPER's timeline is that it can equivocate, *i.e.,* present alternate logs to the public and to some selected targets. CT [163] and CONIKS [178] propose the use of auditors that try to detect such equivocations. In these systems, the authorities maintaining tamper-evident logs post a summary of their log in the form of a Merkle Tree Root (MTR). Auditors collect these MTRs and gossip them trying to retroactively detect inconsistencies. Although this approach may work when auditors are well connected with the targeted users, there is no guarantee that an equivocated MTR will propagate from the target user to the auditors and even if it does, it might still be too late for the targeted user.

In our next step towards SKIPPER, we tackle the problem of such stealthy equivocation, by employing collective witnessing via a cothority. This approach provides pro-active security as no client will trust the claims of an authority maintaining a timeline, unless this timeline is collectively signed and approved by a diverse set of witnesses. Specifically, we make SKIPPER's log cothority-controlled hash chains that create a public history of each timeline. When a timeline update is announced by an authority, the cothority checks that the update is semantically correct depending on the application (*e.g.,* the software update is signed by the authoritative developers [195]) and that there is no fork in the hash-chain (*i.e.,* that the parent hash-block is the last one publicly logged and that there is no other hash-block with the same parent) and runs BFT-CoSi [149] to create a new collective signature. This approach provides an efficient and secure mechanism for timeline extension and moreover allows auditing the actions of all cothority members since malicious nodes can be detected and excluded quickly. The only way to convince the client to accept an equivocated update is to submit it to the

cothority for approval and public logging. Hence, it is not possible for the adversary to sign an equivocated update and keep it "off the public record".

5.4.3 Evolution of Authoritative Keys

So far, we have assumed that cothority keys are static, hence clients who verify (individual or collective) signatures need not rely on centralized intermediaries such as CAs to retrieve those public keys. This assumption is unrealistic, however, as it makes a compromise of a key only a matter of time. Collective signing exacerbates this problem, because for both maximum independence and administrative manageability, witnesses' keys might need to rotate on different schedules. To lift this assumption without relying on centralized CAs, we construct a decentralized mechanism for a trust delegation that enables the evolution of the keys. As a result, developers and cothorities can change, when necessary, their signing keys and create a moving target for an attacker, and the cothority becomes more robust to churn.

To implement this trust delegation mechanism, we introduce forward-linked blockchains. A forward-linked blockchain is a blockchain-like data structure where (a) every block represents the set of authoritative keys representing a certain entity (*e.g.,* the public keys of a cothority) and (b) has forward links that delegate trust from an old representation to a new one, enabling anyone to crawl the chain forward and get up-to-date.

Forward links result in two challenges: first, forward-links can only be added retroactively to blocks of the log, as at the time of block creation, future blocks do not yet exist. Second, for the creation of forward-links, we cannot rely on cryptographic hashes as in the case of back-links, since it would result in a circular dependency between the forward-link of the current and the back-link of the next block. To solve both problems we employ digital signatures. An authority that wants to evolve, *i.e.,* rotate its key, creates a new block containing the new key and a back-link to the current head. Afterwards, it signs the new block with its current key thereby creating a forward-link from the current to the new block which makes the latter also the new head of the chain. That way, the authority delegates the trust from its old key to the new one and a client following the chain in the forward direction can easily verify each step it is taking.

5.4.4 Skipchains

Finally, we extend the above design to provide efficient (*i.e.,* logarithmic) navigability both forward and backward in time via the full skipchain structure we detail here.

Skipchains are authenticated data structures that combine ideas from blockchains [148] and skiplists [207, 190]. Skipchains enable clients (1) to securely traverse the timeline in both forward and backward directions and (2) to efficiently traverse short or long distances by employing multi-hop links. Backward links are cryptographic hashes of past blocks, as in regular blockchains. Forward links are cryptographic signatures of future blocks, which are added retroactively when the target block appears.

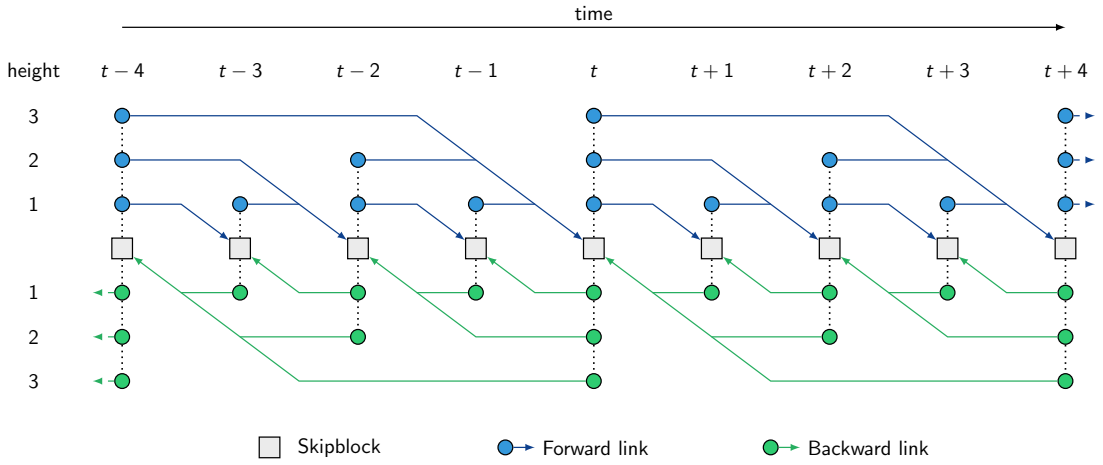

Figure 5.2 – A deterministic skipchain $\mathscr{S}_2^3$

We distinguish *randomized* and *deterministic* skipchains, which differ in the way the lengths of multi-hop links are determined. The link length is tied to the height parameter of a block that is computed during block creation, either randomly in randomized skipchains or via a fixed formula in deterministic skipchains. In both approaches, skipchains enable logarithmic-cost timeline traversal, both forward and backward.

Design We denote a skipchain by $\mathscr{S}_b^h$ where $h \geq 1$ and $b > 0$ are called *skipheight* and *skipbasis*, respectively. If $0 < b < 1$ we call the skipchain randomized; and if $b \geq 1$ (b integer), we call it deterministic. The elements of a skipchain are *skipblocks* $\mathscr{B}_t = (\mathsf{id}_t, h_t, D_t, B_t, F_t)$ where $t \geq 0$ is the block index. The variables id_t, h_t, D_t, B_t, and F_t denote block identifier, block height, payload data, list of backward links, and list of forward links, respectively. Both B_t and F_t can store exactly h_t links and a reference at index $0 \leq i \leq h_t - 1$ in B_t (F_t) points to the last (next) block in the timeline having at least height $i + 1$. For deterministic skipchains this block is $\mathscr{B}_{t-j}$ ($\mathscr{B}_{t+j}$) where $j = b^i$.

The concrete value of h_t is determined by the dependency of the skipchain's type: if $\mathscr{S}_b^h$ is randomized, then a coin, with probability b to land on heads, is repeatedly flipped. Once it lands on tails, we set $h_t = \min\{m, h\}$ where m denotes the number of times it landed on heads up to this point. If $\mathscr{S}_b^h$ is deterministic, we set

$$h_t = \max\{i : 0 \leq i \leq h \wedge 0 \equiv t \bmod b^{i-1}\}\,.$$

Figure 5.2 illustrates a simple deterministic skipchain.

During the creation of a block, its identifier is set to the (cryptographic) hash of D_t and B_t, both known at this point, *i.e.*, $\mathsf{id}_t = \mathsf{H}(D_t, B_t)$. For a backward link from $\mathscr{B}_t$ to $\mathscr{B}_{t-j}$, we simply store id_{t-j} at index i in B_t. This works as in regular blockchains but with the difference that

links can point to blocks further back in the timeline.

Forward links [148], are added retroactively to blocks in the log as digital (multi-)signatures. For a forward link from $\mathscr{B}_t$ to $\mathscr{B}_{t+j}$, we store the cryptographic signature $\langle \mathrm{id}_{t+j} \rangle_{E_t}$ at index i in F_t where E_t denotes the entity (possibly a decentralized collective such as a BFT-CoSi cothority [149, 148, 243]) that represents the head of trust of the system during time step t. To create the required signatures for the forward links until all slots in F_t are full, in particular, E_t must "stay alive" and watch the head of the skipchain. Once this is achieved, the job of E_t is done and it ceases to exist.

5.4.5 Useful Properties and Applications

Skipchains provide a framework for timeline tracking, which can be useful in domains such as cryptocurrencies [149, 191, 150], key-management [148, 178], certificate tracking [163, 2] or, in general, for membership evolution in decentralized systems [242, 243]. Beyond the standard properties of blockchains, skipchains offer the following two useful features.

First, skipchains enable clients to securely and efficiently traverse arbitrarily long timelines, both forward and backward from any reference point. If the client has the correct hash of an existing block and wants to obtain a future or past block in the timeline from an untrusted source (such as a software-update server or a nearby peer), to cryptographically validate the target block (and all links leading to it), the client needs to download only a logarithmic number of additional, intermediate blocks.

Secondly, suppose two resource-constrained clients have two reference points on a skipchain, but have no access to a database containing the full skipchain, *e.g.*, clients exchanging peer-to-peer software updates while disconnected from any central update server. Provided these clients have cached a logarithmic number of additional blocks with their respective reference points – specifically the reference points' next and prior blocks at each level – then the two clients have all the information they need to cryptographically validate each others' reference points. For software updates, forward validation is important when an out-of-date client obtains a newer update from a peer. Reverse validation (via hashes) is useful for secure version rollback, or in other applications, such as efficiently verifying a historical payment on a skipchain for a cryptocurrency.

5.4.6 Security Considerations for Skipchains

We consider forward-links to be more "fragile" than back-links from a security perspective, for several reasons. If any single forward-link is completely compromised, then potentially so is any client that tries to follow that link. For example, if an attacker can forge a forward-link – because the attacker holds a threshold of colluding participants in a collective signing group – then the attacker can forge an alternate "future timeline" and direct out-of-date clients down that forged timeline. Further, such an attacker can potentially equivocate and provide different

forged timelines to different clients. In addition, the cryptographic foundations of current digital signature schemes may be more susceptible than cryptographic hash functions to attacks by quantum computers in the longer term. Thus, even with collective signing, it is preferable to rely more on back-links than forward-links whenever possible.

These properties create a need to balance between efficiency and security considerations for "long" forward-links. Long forward-links are better for allowing extremely out-of-date clients who have been offline for a long time to catch up quickly, but their downside is that the (collective) holders of the private keys corresponding to the starting point for the forward-link must hold and protect their secret keys for a longer time period. Thus, we suggest imposing a maximum time period that forward-links are allowed to cover (and hence a maximum lifetime for the secret keys needed to sign them): a one-year forward-skip limit seems likely to be reasonable for example. Any client several years out of date must then follow forward-links linearly one year at a time, but historically, clients that are more than 5–10 years out-of-date may stop working anyway for unrelated technological obsolescence reasons such as incompatible software or protocols.

5.5 Multi-level Relationships

In this Section, we explore the different timelines that SKIPPER models, which can be layered to combine usability, security, and performance.

5.5.1 Multi-level Service Timelines

The need for secure timeline tracking exists in multiple administrative timescales to allow for different trade-offs between usability, security, and performance. Specifically the most-critical secret keys exist for long timescales, should only be used rarely and be kept offline. while lower valued keys can be online to enable automatic signing; these lower-valued keys are at a higher risk of compromise, but the impact of the attacks is low, detectable and recoverable.

We propose three useful and complementary timeline roles, namely (a) root-of-trust timeline, (b) control/configuration timeline, and (c) data/transaction timeline. Finally, we introduce a timestamping role that the configuration timeline fulfills which protects against freeze attacks [217]. A freeze attack happens when an adversary first eclipses all network connections from a target client and then serves him stale data. For example, he can send him an old configuration block which he (the adversary) managed to compromise after a long time period.

Root-of-Trust Timeline The Internet is an inherently trustless and hostile environment. Any system that wants to operate securely in such a setting, needs to provide a certain base level of trust, from where a client can either securely request a service or start its navigation and reach the correct point of service over a previously established chain of trust.

Due to its fundamental security role, the root of trust is obviously the most critical one from a security perspective and has the highest value to attackers. The root signing keys are often very long-lived and should receive a high level of protection. This might include storing them offline in a secured environment (*e.g.,* a Yubikey) and minimizing the possibility of exposure to attackers by only using them when absolutely necessary. Such special situations might include events like the need for revocation and replacement of compromised keys. Many systems, like DNSSEC [87, 2] or X.509 [66], provide such roots of trust by anchoring. Another option is to provide new trusted roots through software updates [82], however, in both situations, the update of the root is hard especially if the previous anchor is compromised.

Control/Configuration Timeline Due to the necessity of securing the root of trust, it cannot be assumed that the root level service is permanently available. Hence, there is usually an online representative of the root layer, which we call the control layer, that has a more restricted set of permissions but that is able to serve clients and monitor applications. As a consequence, the signing keys need to be stored online which puts them at a higher risk of being compromised by attackers but on the other hand, allows to efficiently serve clients. The life span of epochs in the control layer is usually shorter than of those in the root layer, for example, due to the need for key rotation or participation churn. An example of a control-layer timeline is the key-block blockchain of ByzCoin where the miners delegate trust to the next committee by collectively signing the new valid key-block.

Data/Transaction Timeline These timelines may or may not hold authoritative keys depending on the application. For example, it may be a blockchain that stores transactions and/or data such as the micro-block blockchain of ByzCoin [149] or the transparent log of CONIKS and CT. On the other hand, there are applications where the identity management is strongly integrated into the application and trust delegation is required even at this low-level. One such application is Chainiac [195], where the data timeline does hold the keys of the developers of a software update and data-blocks are doubly linked. Another is our SSH-key management described in Section 5.6

Timestamp Role of Control Timeline Finally, the TIME role provides a timestamp service that informs clients of the latest version of a package, within a coarse-grained time interval. Every TIME block contains a wall-clock timestamp and a hash of the latest release. The CONFIG leader creates this block when a new DATA skipblock is co-signed, or every hour if nothing happens. Before signing it off, the rest of the independent servers check that the hash inside the timestamp is correct and that the time indicated is sufficiently close to their clocks (*e.g.,* within five minutes). From an absence of fresh TIME updates and provided that clients have an approximately accurate notion of the current time[1], the clients can then detect freeze attacks.

[1] Protecting the client's notion of time is an important but orthogonal problem [174], solvable using a timestamping service with collectively-signed proofs-of-freshness, as in CoSi [243].

5.5.2 Multi-Layer Trust Delegation in SKIPPER

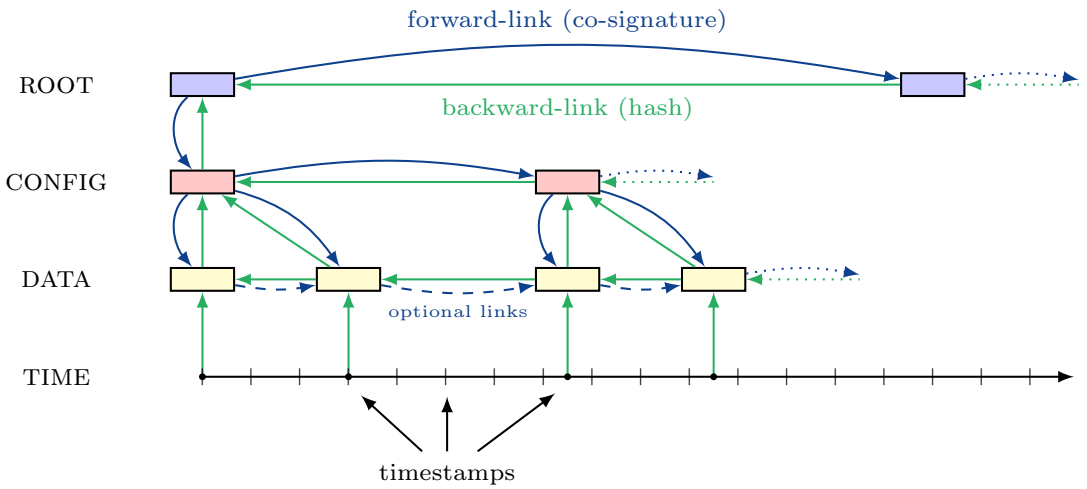

Figure 5.3 – Trust delegation in SKIPPER

SKIPPER is a trust-delegation framework which exhibits a three-tier architecture where ROOT denotes the highest, CONFIG the middle, and DATA the lowest level. Layers ROOT and CONFIG are used to securely track service configurations while the DATA layer is used to track concrete application data. This might again be service configurations but could also be information such as transactions of a cryptocurrency. Each level consists of at least one skipchain ensuring "horizontal" trust-delegation. The ROOT layer usually has just a single skipchain, while layers CONFIG and DATA might have multiple ones that exist next to each other. Vertical trust delegation is done by signing certain skipblocks which happens exclusively in a top-down approach, *i.e.,* from ROOT over CONFIG to DATA, and never from a lower to a higher level. In other words, a particular layer is allowed to sign skipblocks on its own or on lower layers but never on a higher one. A "vertical" downward (upward) link is a cryptographic signature (hash) similarly to a forward (backward) link in a skipchain. Since the ROOT layer is assumed to change only very slowly its respective skipchain has a height of $h = 1$ to achieve maximum security. The CONFIG layer, however, will often change rapidly and thus might better utilize skipchains of height $h > 1$. Figure 5.3 gives an overview of trust delegation in SKIPPER.

5.6 Prototype Implementation

In this section, we describe concrete applications that we have implemented using the SKIPPER framework.

5.6.1 SKIPPER Implementation

We have built and evaluated a working prototype of SKIPPER by implementing a new protocol based on the BFT algorithm of ByzCoin [149] and using it as a lower layer for the creation of skipchains. The implementation which is written in Go [118] and is available at Github. The SKIPPER prototype currently implements a full cothority and multiple layers can be concurrently deployed, connected together with vertical downward links as described earlier. We evaluated the SKIPPER implementation with Schnorr signatures implemented on the Ed25519 curve [27].

Subsequent subsections now discuss particular experimental prototype applications we have implemented and experimented with using this prototype implementation of SKIPPER.

5.6.2 Software Updates with SKIPPER

We explore how skipchains may be useful in the context of securing software updates. Specifically, we integrate skipchains into Diplomat [158], a security system that is designed to be resilient to an attacker that compromises the software repository. Diplomat has three roles of particular interest for this work, the timestamp role (which knows the timestamp of the freshest metadata), the snapshot role (which knows the versions of the latest packages), and the root role (which establishes and revokes trust). We will show how integration with SKIPPER improves the security of Diplomat.

Skipchain on the Root Role

The root of trust in Diplomat is a set of keys that collectively sign the root metadata. These keys are trusted to delegate and revoke trust to other roles in the system. When one of the root keys needs to be replaced (perhaps due to a compromise), a new Skipblock is created, containing the new set of keys (that might overlap with the previous), and signed by all the keys. This effectively creates new root metadata and updates the set of trusted keys. This way the root role achieves a strongest-link security since an attacker with all but one keys is unable to change the root-of-trust and gain full control. Skipchains enable secure tracking of the root-role from an out-of-date client and guarantee a linear timeline. This enables the client to detect any attacks where the attacker got hold of the root role, installed malware and then returned control to the correct root to avoid detection.

SKIPPER in Timestamp and Snapshot metadata

The timestamp role in Diplomat has the highest risk of compromise of any role. This is because it is stored online on the repository and is even sometimes given to mirrors. Thus in Diplomat, this role can be compromised if the weakest party possessing the key is compromised. The snapshot role is also an online role that is stored on the repository. It informs the user of what

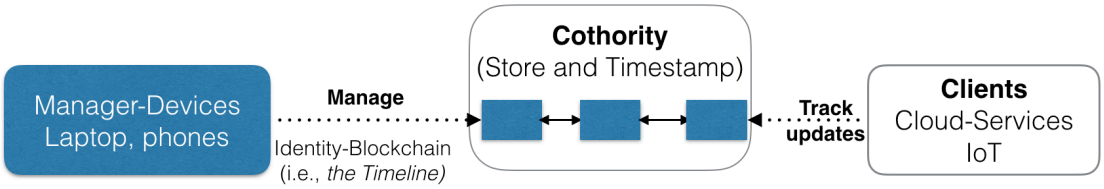

Figure 5.4 – SSH Management with SKIPPER. System Architecture

version of packages is the latest, but does not sign the packages themselves. Compromising just the Timestamp role leads to freeze [217] attacks while additional compromise of the Snapshot role can lead to mix-and-match attacks [217].

Securing the online snapshot and timestamp roles with skipchains greatly improves resilience to compromise and adds anti-equivocation properties. An attacker would need to compromise a threshold of keys in a short timeframe across a wide array of servers to compromise a key. Furthermore, the use of multi-layered skipchains means that the impact of a compromise is minimized, since the chaining of updates enables quick detection and the multiple timeline layering will allow for recovery, since the CONFIG-skipchain can effectively route around the compromised part of the DATA-skipchain, and this way protect the clients, from following the compromised path.

SKIPPER can be deployed as a cothority that only allows one linear timeline of timestamps. This way even if the timestamping and snapshot keys are compromised the attacker will be unable to get two conflicting snapshots signed, one for the public and one specifically tailored for the victim. These security properties can be provided with minimal overhead thanks to higher lever skiplinks as we can see in Section 5.7.2.

5.6.3 SSH-based Distributed User Identities

As an example of the identity-tracking we built an application to manage automatic login to servers using SSH. Figure 5.4 illustrates the architecture of the system.

The *Manager-Devices* are devices that the user can physically access, like laptops, and phones. They are authoritative for managing the user's identity by proposing new data to be included in the blocks. In our use-case, this data consists of public SSH keys. However, the managers are not always available and/or accessible, thus they cannot be used to serve new blocks to the clients, or manage the distributed coordination. To mitigate this problem we assume the existence of publicly reachable servers forming a cothority [243], that provides blockchain management as a service. Any potential user can request his keys to be tracked by the service by generating a separate DATA-timeline where user stores and manages his identity. The DATA-skipblocks consist of the actual list of authoritative keys of the user. If a device is lost the user creates a new list where the lost key is missing, which leads to the generation of a new DATA-skipblock. The same process is followed if a new device is introduced, or the user decides

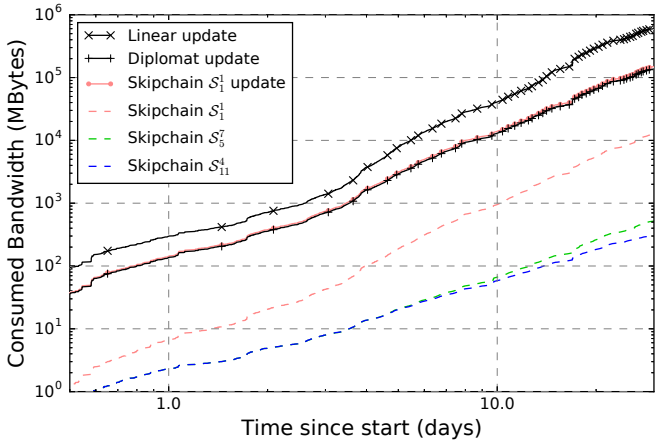

Figure 5.5 – Communication cost for different frameworks

to rotate his keys. SKIPPER simplifies the rotation and revocation of SSH-keys by enabling automatic secure tracking and updating of the `authorized_keys`. The security is increased by requiring a threshold number of identities to sign any update. This security parameter makes it impossible for an attacker who has less than the threshold of devices in his control to tamper with the identity-list of the user and gain full control.

5.7 Experimental Evaluation

In this section, we experimentally evaluate our SKIPPER prototype. The main question we answer is whether SKIPPER is usable in practice without incurring large overheads.

5.7.1 Experimental Methodology

In the experiments we used real-life data from the PyPI package repository [208]. The data represented snapshots of the repository of about 58,000 packages. There were 11,000 snapshots over a period of 30 days. Additionally, we had 1.5 million update-requests from 400,000 clients during the same 30-day period. Using this data, we implemented a simulation in Ruby to compare different bandwidth usages.

5.7.2 Skipchain Effect on PyPI Communication Cost

To evaluate the effect on communication cost of using skipchains for update verification, we compare it with two other scenarios using data from the PyPI package repository. The scenarios are as follows:

1. **Linear update:** When a client requests an update, she downloads all the diffs between snapshots, starting from her last update to the most recent one. This way she validates

every step and tracks the timeline herself.

2. **Diplomat:** The client only downloads the diff between her last update and the latest update available. This configuration does not provide secure timeline tracking and incurs the lowest possible bandwidth overhead.

3. **Skipchain S_1^1:** The scenario is as in Diplomat, but every skipblock is also sent to prove the correctness of the current update. The skipchains add security to the snapshots by signing it and by enabling users to efficiently track changes in the signers.

The results over the 30-day data are presented in Figure 5.5. The straight lines correspond to the aforementioned scenarios. Linear updates increase the communication cost since the cumulative updates between two snapshots can contain different updates, which are only transferred once, of the same package, as in the case of Diplomat or skipchains. As can be seen, the communication costs for Diplomat and skipchain are similar, even in the worst case where a skipchain has height-1 only, which corresponds to a simple double-linked list.

To further investigate the best parameters of the skipchain, we plotted only the skipchain overhead using the same data. In Figure 5.5, the dashed lines show the additional communication cost for different skipchain parameters. We observe that a skipchain with height > 1 can reduce by a factor of 15 the communication cost for proving the validity of a snapshot. Using the base 5 for the skipchain can further reduce the communication cost by another factor of 2.

5.7.3 SSH-based User Identities

We implemented a simple SSH key-management system as described in Section 5.6.3. A manager-device proposes a change to the identity-blockchain of the user which is transmitted to the other manager-devices and has to be confirmed by a threshold of them. The clients have a list of identity-blockchains that they track. In the case where a client missed several updates, he will receive all blocks necessary to prove that the timeline he is tracking is legitimate.

Every manager-device that wants to accept a change needs to transmit a confirmation signature on the update which is about 64 bytes. Once the cothority has verified the manger-devices' signatures it appends a new block on the blockchains that has an overhead of 256 bytes for the hash, the signature, and other configuration-fields for the blockchain. When a client downloads the new list it also has to download that overhead and then needs to verify that the signatures of the manager-devices correct. We have also implemented the option to delegate the individual signature verification to the storage cothority. In this case, the cothority verifies the manager-devices' signatures and then collectively signs the block. This verification of the collective signatures uses only the aggregate public key of the cothority and as such is lightweight, even in the case of a large number of cothority-servers, making the system suitable even for resource-constrained devices (*e.g.,* in IoT).

5.8 Conclusion

We described SKIPPER, a decentralized relationship-keeping system that leverages blockchain technology and scalable byzantine fault-tolerant collective signing to enable clients to asynchronously but securely track the updates of decentralized and distributed systems. SKIPPER is built on three distinct layers of skipchains to provide security, usability, and performance. Clients need not verify all the previous updates to retrieve the information they need, but can walk and jump through the timeline to quickly verify the information they want without relying on intermediaries. On simulations we showed that SKIPPER can demonstrably increase the security of Diplomat, a software update framework, while introducing minimal overhead as well as enable secure and usable SSH-key management.

Part III
Private and Horizontally Scalable Distributed Ledgers

6 OMNILEDGER: A Secure, Scale-Out, Decentralized Ledger via Sharding

6.1 Introduction

The scalability of distributed ledgers (DLs), in both total transaction volume and the number of independent participants involved in processing them, is a major challenge to their mainstream adoption, especially when weighted against security and decentralization challenges. Many approaches exhibit different security and performance trade-offs [47, 10, 95, 149, 205]. Replacing the Nakamoto consensus [191] with PBFT [55], for example, can increase throughput while decreasing transaction commit latency [3, 149]. These approaches still require all *validators* or consensus group members to redundantly validate and process all transactions, hence the system's total transaction processing capacity does not increase with added participants, and, in fact, gradually decreases due to increased coordination overheads.

The proven and obvious approach to building *scale-out* databases, whose capacity scales horizontally with the number of participants, is by *sharding* [67], or partitioning the state into multiple shards that are handled in parallel by different subsets of participating validators. Sharding could benefit DLs [70] by reducing the transaction processing load on each validator and by increasing the system's total processing capacity proportionally with the number of participants. Existing proposals for sharded DLs, however, forfeit permissionless decentralization [74], introduce new security assumptions, and/or trade performance for security [173], as illustrated in Figure 6.1 and explored in detail in Section 6.2.

In this chapter we introduce OMNILEDGER, the first DL architecture that provides scale-out transaction processing capacity competitive with centralized payment-processing systems, such as Visa, without compromising security or support for permissionless decentralization. To achieve this goal, OMNILEDGER faces three key correctness and security challenges. First, OMNILEDGER must choose statistically representative groups of validators periodically via permissionless Sybil-attack-resistant foundations such as proof-of-work [191, 198, 149] or proof-of-stake [116]. Second, OMNILEDGER must ensure a negligible probability that any shard is compromised across the (long-term) system lifetime via periodically (re-)forming shards (subsets of validators to record state and process transactions), that are both sufficiently

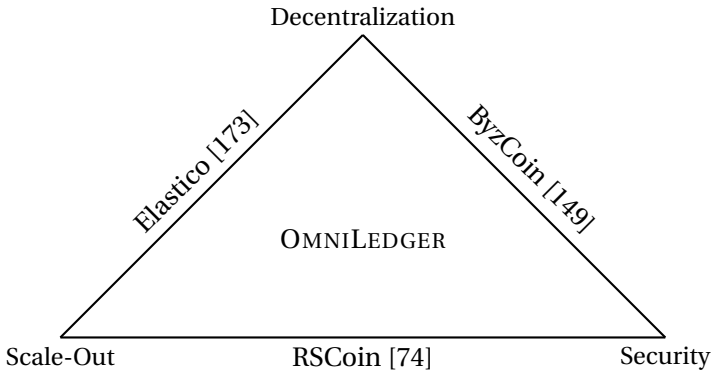

Figure 6.1 – Trade-offs in current DL systems.

large and bias-resistant. Third, OMNILEDGER must correctly and atomically handle *cross-shard transactions*, or transactions that affect the ledger state held by two or more distinct shards.

To choose representative validators via proof-of-work, OMNILEDGER builds on ByzCoin (Chapter 3) and Hybrid Consensus [198], using a sliding window of recent proof-of-work block miners as its validator set. To support the more power-efficient alternative of apportioning consensus group membership based on directly invested stake rather than work, OMNILEDGER builds on Algorand [116], running a public randomness or cryptographic sortition protocol within a prior validator group to pick a subsequent validator group from the current stakeholder distribution defined in the ledger. To ensure that this sampling of representative validators is both scalable and strongly bias-resistant, OMNILEDGER uses RandHound (Chapter 4), a protocol that serves this purpose under standard t-of-n threshold assumptions.

Appropriate use of RandHound provides the basis by which OMNILEDGER addresses the second key security challenge of securely assigning validators to shards, and of periodically rotating these assignments as the set of validators evolves. OMNILEDGER chooses shards large enough, based on the analysis in Section 6.6, to ensure a negligible probability that any shard is ever compromised, even across years of operation.

Finally, to ensure that transactions either commit or abort atomically even when they affect state distributed across multiple shards (*e.g.*, several cryptocurrency accounts), OMNILEDGER introduces Atomix, a two-phase client-driven "lock/unlock" protocol that ensures that clients can either fully commit a transaction across shards, or obtain *rejection proofs* to abort and unlock state affected by partially completed transactions.

Besides addressing the above key security challenges, OMNILEDGER also introduces several performance and scalability refinements we found to be instrumental in achieving its usability goals. OMNILEDGER's consensus protocol, *ByzCoinX*, enhances the PBFT-based consensus in ByzCoin [149] to preserve performance under Byzantine denial-of-service (DoS) attacks, by adopting a more robust group communication pattern. To help new or long-offline miners catch up to the current ledger state without having to download the entire history,

OMNILEDGER adapts classic distributed checkpointing principles [92] to produce consistent, *state blocks* periodically.

Finally, to minimize transaction latency in common cases such as low-value payments, OMNILEDGER supports optional *trust-but-verify* validation in which a first small tier of validators processes the transactions quickly and then hands them over to a second larger, hence slower, tier that re-verifies the correctness of the first tier transactions and ensures long-term security. This two-level approach ensures that any misbehavior within the first tier is detected within minutes, and can be strongly disincentivized through recourse such as loss of deposits. Clients can wait for both tiers to process high-value transactions for maximum security or just wait for the first tier to process low-value transactions.

To evaluate OMNILEDGER, we implemented a prototype in Go on commodity servers (12-core VMs on Deterlab). Our experimental results show that OMNILEDGER scales linearly in the number of validators, yielding a throughput of 6,000 transactions per second with a 10-second consensus latency (for 1800 widely-distributed hosts, of which 12.5% are malicious). Furthermore, deploying OMNILEDGER with two-level, trust-but-verify validation provides a throughput of 2,250 tps with a four-second first-tier latency under a 25% adversary. Finally, a Bitcoin validator with a month-long stale view of the state incurs 40% of the bandwidth, due to state blocks.

In summary, this chapter makes the following contributions:

- We introduce the first DL architecture that provides horizontal scaling without compromising either long-term security or permissionless decentralization.
- We introduce Atomix, an Atomic Commit protocol, to commit transactions atomically across shards.
- We introduce ByzCoinX, a BFT consensus protocol that increases performance and robustness to DoS attacks.
- We introduce state blocks, that are deployed along OMNILEDGER to minimize storage and update overhead.
- We introduce two-tier trust-but-verify processing to minimize the latency of low-value transactions.

6.2 Preliminaries

6.2.1 Transaction Processing and the UTXO model

Distributed ledgers derive current system state from a *blockchain*, or a sequence of totally ordered blocks that contain transactions. OMNILEDGER adopts the *unspent transaction output*

(UTXO) model to represent ledger state, due to its simplicity and parallelizability. In this model, the outputs of a transaction create new UTXOs (and assign them credits), and inputs completely "spend" existing UTXOs. During bootstrapping, new (full) nodes crawl the entire distributed ledger and build a database of valid UTXOs needed to subsequently decide whether a new block can be accepted. The UTXO model was introduced by Bitcoin [191] but has been widely adopted by other distributed ledger systems.

6.2.2 Prior Sharded Ledgers: Elastico

OMNILEDGER builds closely on Elastico [173], which previously explored sharding in permissionless ledgers. In every round, Elastico uses the least-significant bits of the PoW hash to distribute miners to different shards. After this setup, every shard runs PBFT [55] to reach consensus, and a leader shard verifies all the signatures and creates a global block.

OMNILEDGER addresses several challenges that Elastico leaves unsolved. First, Elastico's relatively small shards (*e.g.,* 100 validators per shard in experiments) yield a high failure-probability of 2.76%[1] per shard per block under a 25% adversary, which cannot safely be relaxed in a PoW system. For 16 shards, the failure probability is 97% over only 6 epochs. Second, Elastico's shard selection is not strongly bias-resistant, as miners can selectively discard PoWs to bias results [41]. Third, Elastico does not ensure transaction atomicity across shards, leaving funds in one shard locked forever if another shard rejects the transaction. Fourth, the validators constantly switch shards, forcing themselves to store the global state, which can hinder performance but provides stronger guarantees against adaptive adversaries. Finally, the latency of transaction commitment is comparable to Bitcoin ($\approx$ 10 min.), which is far from OMNILEDGER's usability goals.

6.2.3 Sybil-Resistant Identities

Unlike permissioned blockchains [74], where the validators are known, permissionless blockchains need to deal with the potential of Sybil attacks [84] to remain secure. Bitcoin [191] suggested the use of Proof-of-Work (PoW), where validators (aka miners) create a valid block by performing an expensive computation (iterating through a nonce and trying to brute-force a hash of a block's header such that it has a certain number of leading zeros). Bitcoin-NG [95] uses this PoW technique to enable a Sybil-resistant generation of identities. There are certain issues associated with PoW, such as the waste of electricity [79] and the fact that it causes recentralization [134] to mining pools. Other approaches for establishing Sybil-resistant identities such as Proof-of-Stake (PoS) [116], Proof-of-Burn (PoB) [247] or Proof-of-Personhood [43] overcome PoW's problems and are compatible with ByzCoin's identity (key-block) blockchain, and in turn with OMNILEDGER.

[1] Cumulative binomial distribution ($P = 0.25$, $N = 100$, $X \geq 34$)

6.3 System Overview

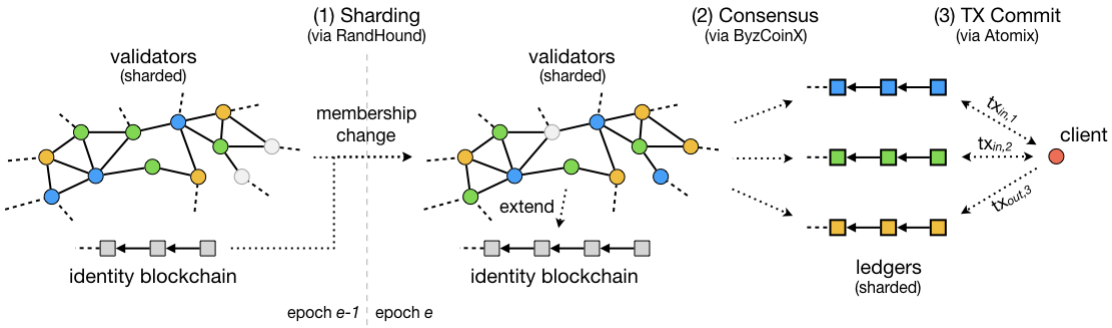

Figure 6.2 – OMNILEDGER architecture overview: At the beginning of an epoch e, all validators (shard membership is visualized through the different colors) (1) run RandHound to re-assign randomly a certain threshold of validators to new shards and assign new validators who registered to the identity blockchain in epoch $e-1$. Validators ensure (2) consistency of the shards' ledgers via ByzCoinX while clients ensure (3) consistency of their cross-shard transactions via Atomix (here the client spends inputs from shards 1 and 2 and outputs to shard 3).

This section presents the system, network and threat models, the design goals, and a roadmap towards OMNILEDGER that begins with a strawman design.

6.3.1 System Model

We assume that there are n *validators* who process transactions and ensure the consistency of the system's state. Each validator i has a public / private key pair $(\mathsf{pk}_i, \mathsf{sk}_i)$, and we often identify i by pk_i. Validators are evenly distributed across m *shards*. We assume that the configuration parameters of a shard j are summarized in a *shard-policy file*. We denote by an *epoch* e the fixed time (*e.g.,* a day) between global reconfiguration events where a new assignment of validators to shards is computed. The time during an epoch is counted in *rounds* r that do not have to be consistent between different shards. During each round, each shard processes transactions collected from clients. We assume that validators can establish identities through any Sybil-attack-resistant mechanism and commit them to the identity blockchain; to participate in epoch e validators have to register in epoch $e-1$. These identities are added into an identity blockchain as described in Chapter 3.

6.3.2 Network Model

For the underlying network, we make the same assumption as prior work [173, 191]. Specifically, we assume that (a) the network graph of honest validators is well connected and that (b) the communication channels between honest validators are synchronous, *i.e.,* that if an honest validator broadcasts a message, then all honest validators receive the message within a known maximum delay of Δ [199]. However, as Δ is on the scale of minutes, we cannot use it

within epochs as we target latencies of seconds. Thus, all protocols inside one epoch use the partially synchronous model [55] with optimistic, exponentially increasing time-outs, whereas Δ is used for slow operations such as identity creation and shard assignment.

6.3.3 Threat Model

We denote the number of Byzantine validators by f and assume, that $n = 4f$, *i.e.,* at most 25%[2] of the validators can be malicious at any given moment, which is similar to prior DL's [95, 149, 173]. These malicious nodes can behave arbitrarily, *e.g.,* they might refuse to participate or collude to attack the system. The remaining validators are honest and faithfully follow the protocol. We further assume that the adversary is *mildly adaptive* [173] on the order of epochs, *i.e.,* he can try to corrupt validators, but it takes some time for such corruption attempts to actually take effect.

We further assume that the adversary is computationally bounded, that cryptographic primitives are secure, and that the computational Diffie-Hellman problem is hard.

6.3.4 System Goals

OMNILEDGER has the following primary goals with respect to decentralization, security, and scalability.

1. **Full decentralization.** OMNILEDGER does not have any single points of failure (such as trusted third parties).
2. **Shard robustness.** Each shard correctly and continuously processes transactions assigned to it.
3. **Secure transactions.** Transactions are committed atomically or eventually aborted, both within and across shards.
4. **Scale-out.** The expected throughput of OMNILEDGER increases linearly in the number of participating validators.
5. **Low storage overhead.** Validators do not need to store the full transaction history but only a periodically computed reference point that summarizes a shard's state.
6. **Low latency.** OMNILEDGER provides low latency for transaction confirmations.

6.3.5 Design Roadmap

This section introduces SLedger, a strawman DL system that we use to outline OMNILEDGER's design. Below we describe one epoch of SLedger and show how it transitions from epoch $e-1$

[2] The system can handle up to 33% $-\epsilon$ with degraded performance.

to epoch e.

We start with the secure validator-assignment to shards. Permitting the validators to choose the shards they want to validate is insecure, as the adversary could focus all his validators in one shard. As a result, we need a source of randomness to ensure that the validators of one shard will be a sample of the overall system and w.h.p. will have the same fraction of malicious nodes. SLedger operates a trusted randomness beacon that broadcasts a random value rnd_e to all participants in each epoch e. Validators, who want to participate in SLedger starting from epoch e, have to first register to a global identity blockchain. They create their identities through a Sybil-attack-resistant mechanism in epoch $e-1$ and broadcast them, together with the respective proofs, on the gossip network at most Δ before epoch $e-1$ ends.

Epoch e begins with a leader, elected using randomness rnd_{e-1}, who requests from the already registered and active validators a (BFT) signature on a block with all identities that have been provably established so far. If at least $\frac{2}{3}$ of these validators endorse the block, it becomes valid, and the leader appends it to the identity blockchain. Afterwards, all registered validators take rnd_e to determine their assignment to one of the SLedger's shards and to bootstrap their internal states from the shards' distributed ledgers. Then, they are ready to start processing transactions using ByzCoin. The random shard-assignment ensures that the ratio between malicious and honest validators in any given shard closely matches the ratio across all validators with high probability.

SLedger already provides similar functionality to OMNILEDGER, but it has several significant security restrictions. First, the randomness beacon is a trusted third party. Second, the system stops processing transactions during the global reconfiguration at the beginning of each epoch until enough validators have bootstrapped their internal states and third, there is no support for cross-shard transactions. SLedger's design also falls short in performance. First, due to ByzCoin's failure handling mechanism, its performance deteriorates when validators fail. Second, validators face high storage and bootstrapping overheads. Finally, SLedger cannot provide real-time confirmation latencies and high throughput.

To address the security challenges, we introduce OMNILEDGER's security design in Section 6.4:

1. In Section 6.4.1, we remove the trusted randomness beacon and show how validators can autonomously perform a secure sharding by using a combination of RandHound and VRF-based leader election via cryptographic sortition.
2. In Section 6.4.2, we show how to securely handle the validator assignment to shards between epochs while maintaining the ability to continuously process transactions.
3. In Section 6.4.3, we present *Atomix*, a novel two-step atomic commit protocol for atomically processing cross-shard transactions in a Byzantine setting.

To deal with the performance challenges, we introduce OMNILEDGER's performance and

usability design in Section 6.5:

4) In Section 6.5.1, we introduce *ByzCoinX*, a variant of ByzCoin, that utilizes more robust communication patterns to efficiently process transactions within shards, even if some of the validators fail, and that resolves dependencies on the transaction level to achieve better block parallelization.

5) In Section 6.5.3, we introduce *state blocks* that summarize the shards' states in an epoch and that enable ledger pruning to reduce storage and bootstrapping costs for validators.

6) In Section 6.5.4, we show how to enable optimistic real-time transaction confirmations without sacrificing security or throughput by utilizing an intra-shard architecture with *trust-but-verify transaction validation*.

A high-level overview of the (security) architecture of OMNILEDGER is illustrated in Figure 6.2.

6.4 OMNILEDGER: Security Design

6.4.1 Sharding via Bias-Resistant Distributed Randomness

To generate a seed for sharding securely without requiring a trusted randomness beacon [74] or binding the protocol to PoW [173], we rely on a distributed randomness generation protocol that is collectively executed by the validators.

We require that the distributed-randomness generation protocol provide unbiasability, unpredictability, third-party verifiability, and scalability. Multiple proposals exist [41, 125, 242]. The first approach relies on Bitcoin, whereas the other two share many parts of the design; we focus on RandHound [242] due to better documentation and open-source implementation.

Because RandHound relies on a leader to orchestrate the protocol run, we need an appropriate mechanism to select one of the validators for this role. If we use a deterministic approach to perform leader election, then an adversary might be able to enforce up to f out of n failures in the worst case by refusing to run the protocol, resulting in up to $\frac{1}{4}n$ failures given our threat model. Hence, the selection mechanism must be unpredictable and unbiasable, which leads to a chicken-and-egg problem as we use RandHound to generate randomness with these properties in the first place. To overcome this predicament, we combine RandHound with a VRF-based leader election algorithm [242, 116].

At the beginning of an epoch e, each validator i computes a ticket $\mathsf{ticket}_{i,e,v} = \mathsf{VRF}_{\mathsf{sk}_i}(\text{“leader”} \parallel \mathsf{config}_e \parallel v)$ where config_e is the configuration containing all properly registered validators of epoch e (as stored in the identity blockchain) and v is a view counter. Validators then gossip these tickets with each other for a time Δ, after which they lock in the lowest-value valid ticket they have seen thus far and accept the corresponding node as the leader of the

RandHound protocol run. If the elected node fails to start RandHound within another Δ, validators consider the current run as failed and ignore this validator for the rest of the epoch, even if he returns later on. In this case, the validators increase the view number to $v+1$ and re-run the lottery. Once the validators have successfully completed a run of RandHound and the leader has broadcast rnd_e together with its correctness proof, each of the n properly registered validators can first verify and then use rnd_e to compute a permutation π_e of $1,\ldots,n$ and subdivide the result into m approximately equally-sized buckets, thereby determining its assignment of nodes to shards.

Security Arguments We make the following observations to informally argue the security of the above approach. Each participant can produce only a single valid ticket per view v in a given epoch e, because the VRF-based leader election starts only after the valid identities have been fixed in the identity blockchain. Furthermore, as the output of a VRF is unpredictable as long as the private key sk_i is kept secret, the tickets of non-colluding nodes, hence the outcome of the lottery is also unpredictable. The synchrony bound Δ guarantees that the ticket of an honest leader is seen by all other honest validators. If the adversary wins the lottery, he can decide either to comply and run the RandHound protocol or to fail, which excludes that particular node from participating for the rest of the epoch.

After a successful run of RandHound, the adversary is the first to learn the randomness, hence the sharding assignment, however, his benefit is minimal. The adversary can again either decide to cooperate and publish the random value or withhold it in the hope of winning the lottery again and obtaining a sharding assignment that fits his agenda better. However, the probability that an adversary wins the lottery a times in a row is upper bounded by the exponentially decreasing term $(f/n)^a$. Thus, after only a few re-runs of the lottery, an honest node wins with high probability and coordinates the sharding. Finally, we remark that an adversary cannot collect random values from multiple runs and then choose the one he likes best as validators accept only the latest random value that matches their view number v.

Breaking the Network Model Although DL protocols that assume a non-static group of validators have similar synchrony assumptions [173, 191], in this section we discuss what can happen if the adversary manages to break them [13]. In such a case we can detect the attack and provide a back-up randomness generation mechanism which is not expected to scale but guarantees safety even in asynchrony.

Given that RandHound guarantees safety without the need for synchrony an adversary manipulates the network can at most slow down any validator he does not control, winning the leadership all the time. However this does not enable the adversary to manipulate RandHound; it just gives him the advantage of being able to restart the protocol if he does not like the random number. This restart will be visible to the network, and the participants can suspect a bias-attempt, when multiple consecutive RandHound rounds start to fail.

OMNILEDGER can provide a "safety valve" mechanism in order to mitigate this problem. When 5 RandHound views fail in a row, which under normal circumstances could happen with less than 1% probability, the validators switch from RandHound to running an asynchronous coin-tossing protocol [49] in order to produce the epoch's randomness. This protocol scales poorly ($O(n^3)$), but it will be run when the network is anyway under attack and liveness is not guaranteed, in which case safety is more important.

6.4.2 Maintaining Operability During Epoch Transitions

Recall that, in each epoch e, SLedger changes the assignments of all n validators to shards, which results in an idle phase during which the system cannot process transactions until enough validators have finished bootstrapping.

To maintain operability during transition phases, OMNILEDGER gradually swaps in new validators to each shard per epoch. This enables the remaining operators to continue providing service (in the honest scenario) to clients while the recently joined validators are bootstrapping. In order to achieve this continued operation, we can swap-out at most $\frac{1}{3}$ of the shard's size ($\approx \frac{n}{m}$), however, the bigger the batch is, the higher the risk gets that the number of remaining honest validators is insufficient to reach consensus and the more stress the bootstrapping of new validators causes to the network.

To balance the chances of a temporary loss of liveness, the shard assignment of validators in OMNILEDGER works as follows. First, we fix a parameter $k < \frac{1}{3}\frac{n}{m}$ specifying the swap-out batch, *i.e.,* the number of validators that are swapped out at a given time. For OMNILEDGER, we decided to work in batches of $k = \log\frac{n}{m}$. Then for each shard j, we derive a seed $\mathsf{H}(j \parallel \mathsf{rnd}_e)$ to compute a permutation $\pi_{j,e}$ of the shard's validators, and we specify the permutation of the batches. We also compute another seed $\mathsf{H}(0 \parallel \mathsf{rnd}_e)$ to permute and scatter the validators who joined in epoch e and to define the order in which they will do so (again, in batches of size k). After defining the random permutations, each batch waits for Δ before starting to bootstrap in order to spread the load on the network. Once a validator is ready, he sends an announcement to the shard's leader who then swaps the validator in.

Security Arguments During the transition phase, we ensure the safety of the BFT consensus in each shard as there are always at least $\frac{2}{3}\frac{n}{m}$ honest validators willing to participate in the consensus within each shard. And, as we use the epoch's randomness rnd_e to pick the permutation of the batches, we keep the shards' configurations a moving target for an adaptive adversary. Finally, as long as there are $\frac{2}{3}\frac{n}{m}$ honest and up-to-date validators, liveness is guaranteed. Whereas if this quorum is breached during a transition (the new batch of honest validators has not yet updated), the liveness is lost only temporarily, until the new validators update.

6.4.3 Cross-Shard Transactions

To enable value transfer between different shards thereby achieving shard interoperability, support for secure cross-shard transactions is crucial in any sharded-ledger system. We expect that the majority of transactions to be cross-shard in the traditional model where UTXOs are randomly assigned to shards for processing [74, 173], see [150].

A simple but inadequate strawman approach to a cross-shard transaction, is to send a transaction concurrently to several shards for processing because some shards might commit the transaction while others might abort. In such a case, the UTXOs at the shard who accepted the transactions are lost as there is no straightforward way to roll back a half-committed transaction, without adding exploitable race conditions.

To address this issue, we propose a novel *Byzantine Shard Atomic Commit (Atomix)* protocol for *atomically* processing transactions across shards, such that each transaction is either committed or eventually aborted. The purpose is to ensure consistency of transactions between shards, to prevent double spending and to prevent unspent funds from being locked forever. In distributed computing, this problem is known as atomic commit [248] and atomic commit protocols [120, 142] are deployed on honest but unreliable processors. Deploying such protocols in OMNILEDGER is unnecessarily complex, because the shards are collectively honest, do not crash infinitely, and run ByzCoin (that provides BFT consensus). Atomix improves the strawman approach with a lock-then-unlock process. We intentionally keep the shards' logic simple and make any direct shard-to-shard communication unnecessary by tasking the client with the responsibility of driving the unlock process while permitting any other party (*e.g.,* validators or even other clients) to fill in for the client if a specific transaction stalls after being submitted for processing.

Atomix uses the UTXO state model, see Section 6.2.1 for an overview, which enables the following simple and efficient three-step protocol, also depicted in Figure 6.3.

1. **Initialize.** A client creates a *cross-shard transaction* (cross-TX for short) whose inputs spend UTXOs of some *input shards* (ISs) and whose outputs create new UTXOs in some *output shards* (OSs). The client gossips the cross-TX and it eventually reaches all ISs.

2. **Lock.** All input shards associated with a given cross-TX proceed as follows. First, to decide whether the inputs can be spent, each IS leader validates the transaction within his shard. If the transaction is valid, the leader marks within the state that the input is spent, logs the transaction in the shard's ledger and gossips a *proof-of-acceptance*, a signed Merkle proof against the block where the transaction is included. If the transaction is rejected, the leader creates an analogous *proof-of-rejection*, where a special bit indicates an acceptance or rejection. The client can use each IS ledger to verify his proofs and that the transaction was indeed locked. After all ISs have processed the lock request, the client holds enough proofs to either commit the transaction or abort it and reclaim any locked funds, but not both.

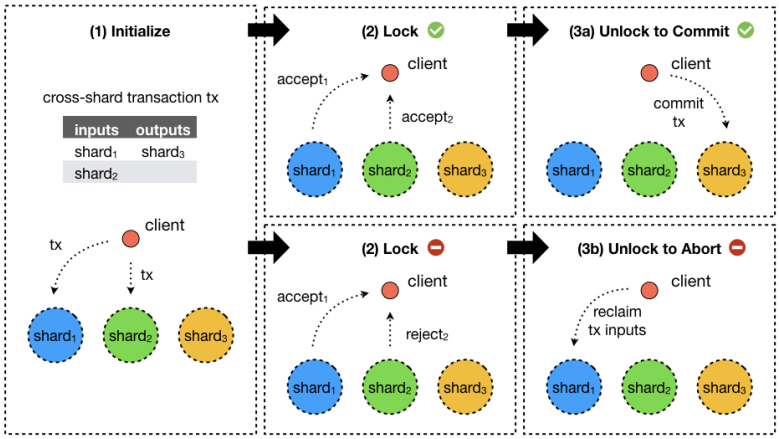

Figure 6.3 – Atomix protocol in OMNILEDGER.

3. **Unlock.** Depending on the outcome of the lock phase, the client is able to either commit or abort his transaction.

   (a) **Unlock to Commit.** If *all* IS leaders issued proofs-of-acceptance, then the respective transaction can be committed. The client (or any other entity such as an IS leader after a time-out) creates and gossips an *unlock-to-commit transaction* that consists of the lock transaction and a proof-of-acceptance for each input UTXO. In turn, each involved OS validates the transaction and includes it in the next block of its ledger in order to update the state and enable the expenditure of the new funds.

   (b) **Unlock to Abort.** If, however, *any one* IS issued a proof-of-rejection, then the transaction cannot be committed and has to abort. In order to reclaim the funds locked in the previous phase, the client (or any other entity) must request the involved ISs to unlock that particular transaction by gossiping an *unlock-to-abort transaction* that includes (at least) one proof-of-rejection for one of the input UTXOs. Upon receiving a request to unlock, the ISs' leaders follow a similar procedure and mark the original UTXOs as spendable again.

Although the focus of OMNILEDGER is on the UTXO model, Atomix can be extended with a locking mechanism for systems where objects are long-lived and hold state (*e.g.,* smart contracts [250]), see Section 6.9.1 for details.

Security Arguments We informally argue the previously stated security properties of Atomix, based on the following observations. Under our assumptions, shards are honest, do not fail, eventually receive all messages and reach BFT consensus. Consequently, (1) all shards always faithfully process valid transactions; (2) if all input shards issue a proof-of-acceptance, then every output shard unlocks to commit; (3) if even one input shard issues a proof-of-rejection, then all input shards unlock to abort; and (4) if even one input shard issues a proof-of-rejection, then no output shard unlocks to commit.

In Atomix, each cross-TX eventually commits or aborts. Based on (1), each input shard returns exactly one response: either a proof-of-acceptance or a proof-of-rejection. Consequently, if a client has the required number of proofs (one per each input UTXO), then the client either only holds proofs-of-acceptance (allowing the transaction to be committed as (2) holds) or not (forcing the transaction to abort as (3) and (4) holds), but not both simultaneously.

In Atomix, no cross-TX can be spent twice. As shown above, cross-shard transactions are atomic and are assigned to specific shards who are solely responsible for them. Based on (1), the assigned shards do not process a transaction twice and no other shard attempts to unlock to commit.

In Atomix, if a transaction cannot be committed, then the locked funds can be reclaimed. If a transaction cannot be committed, then there must exist at least one proof-of-rejection issued by an input shard, therefore (3) must hold. Once all input shards unlock to abort, the funds become available again.

We remark that funds are not automatically reclaimed and a client or other entity must initiate the unlock-to-abort process. Although this approach poses the risk that if a client crashes indefinitely his funds remain locked, it enables a simplified protocol with minimal logic that requires no direct shard-to-shard communication. A client who crashes indefinitely is equivalent to a client who loses his private key, which prevents him from spending the corresponding UTXOs. Furthermore, any entity in the system, for example, a validator in exchange for a fee, can fill in for the client to create an unlock transaction, as all necessary information is gossiped.

To ensure better robustness, we can also assign the shard of the smallest-valued input UTXO to be a coordinator responsible for driving the process of creating unlock transactions. Because a shard's leader might be malicious, $f + 1$ validators of the shard need to send the unlock transaction to guarantee that all transactions are eventually unlocked.

Size of Unlock Transactions In Atomix, the unlock transactions are larger than regular transactions as appropriate proofs for input UTXOs need to be included. OMNILEDGER relies on ByzCoinX (described in Section 6.5.1) for processing transactions within each shard. When the shard's validators reach an agreement on a block that contains committed transactions, they produce a collective signature whose size is independent of the number of validators. This important feature enables us to keep Atomix proofs (and consequently the unlock transactions) short, even though the validity of each transaction is checked against the signed blocks of all input UTXOs. If ByzCoinX did not use collective signatures, the size of unlock transactions would be impractical. For example, for a shard of 100 validators, a collective signature would only be 77 bytes, whereas a regular signature would be 9KB, almost two order's of magnitude larger than the size of a simple transaction (500 bytes).

6.5 Design Refinements for Performance

In this section, we introduce the performance sub-protocols of OMNILEDGER. First, we describe a scalable BFT-consensus called *ByzCoinX* that is more robust and more parallelizable than ByzCoin. Then, we introduce state-blocks that enable fast bootstrapping and decrease storage costs. Finally, we propose an optional trust-but-verify validation step to provide real-time latency for low-risk transactions

6.5.1 Fault Tolerance under Byzantine Faults

The original ByzCoin design offers good scalability, partially due to the usage of a tree communication pattern. Maintaining such communication trees over long time periods can be difficult, as they are quite susceptible to faults. In the event of a failure, ByzCoin falls back on a more robust all-to-all communication pattern, similarly to PBFT. Consequently, the consensus performance deteriorates significantly, which the adversary can exploit to hinder the system's performance.

To achieve better fault tolerance in OMNILEDGER, without resorting to a PBFT-like all-to-all communication pattern, we introduce for ByzCoinX a new communication pattern that trades-off some of ByzCoin's high scalability for robustness, by changing the message propagation mechanism within the consensus group to resemble a two-level tree. During the setup of OMNILEDGER in an epoch, the generated randomness is not only used to assign validators to shards but also to assign them evenly to groups within a shard. The number of groups g, from which the maximum group size can be derived by taking the shard size into account, is specified in the shard policy file. At the beginning of a ByzCoinX roundtrip, the protocol leader randomly selects one of the validators in each group to be the *group leader* responsible for managing communication between the protocol leader and the respective group members. If a group leader does not reply before a predefined timeout, the protocol leader randomly chooses another group member to replace the failed leader. As soon as the protocol leader receives more than $\frac{2}{3}$ of the validators' acceptances, he proceeds to the next phase of the protocol. If the protocol leader fails, all validators initiate a PBFT-like view-change procedure.

6.5.2 Parallelizing Block Commitments

We now show how ByzCoinX parallelizes block commitments in the UTXO model by analyzing and handling dependencies between transactions.

We observe that transactions that do not conflict with each other can be committed in different blocks and consequently can be safely processed in parallel. To identify conflicting transactions, we need to analyze the dependencies that are possible between transactions. Let tx_A and tx_B denote two transactions. Then, there are two cases that need to be carefully handled: (1) both tx_A and tx_B try to spend the same UTXO and (2) a UTXO created at the output of tx_A

is spent at the input of tx_B (or vice versa). To address (1) and maintain consistency, only one of the two tx can be committed. To address (2), tx_A has to be committed to the ledger before tx_B, *i.e.,* tx_B has to be in a block that depends (transitively) on the block containing tx_A. All transactions that do not exhibit these two properties can be processed safely in parallel. In particular, we remark that transactions that credit the same address do not produce a conflict, because they generate different UTXOs

To capture the concurrent processing of blocks, we adopt a *block-based directed acyclic graph (blockDAG)* [167] as a data structure, where every block can have multiple parents. The ByzCoinX protocol leader enforces that each pending block includes only non-conflicting transactions and captures UTXO dependencies by adding the hashes of former blocks (*i.e.,* backpointers) upon which a given block's transactions depend. To decrease the number of such hashes, we remark that UTXO dependencies are transitive, enabling us to relax the requirement that blocks have to capture all UTXO dependencies directly. Instead, a given block can simply add backpointers to a set of blocks, transitively capturing all dependencies.

6.5.3 Shard Ledger Pruning

Now we tackle the issues of an ever-growing ledger and the resulting costly bootstrapping of new validators; this is particularly urgent for high-throughput DL systems. For example, whereas Bitcoin's blockchain grows by ≈ 144 MB per day and has a total size of about 133 GB, next-generation systems with Visa-level throughput (*e.g.,* 4000 tx/sec and 500 B/tx) can easily produce over 150 GB per day.

To reduce the storage and bootstrapping costs for validators (whose shard assignments might change periodically), we introduce *state blocks*, which are similar to stable checkpoints in PBFT [55] and that summarize the entire state of a shard's ledger. To create a state block $\mathsf{sb}_{j,e}$ for shard j in epoch e, the shard's validators execute the following steps: At the end of e, the shard's leader stores the UTXOs in an ordered Merkle tree and puts the Merkle tree's root hash in the header of $\mathsf{sb}_{j,e}$. Afterwards, the validators run consensus on the header of $\mathsf{sb}_{j,e}$ (note that each validator can construct the same ordered Merkle tree for verification) and, if successful, the leader stores the approved header in the shard's ledger making $\mathsf{sb}_{j,e}$ the genesis block of epoch $e+1$. Finally, the body of $\mathsf{sb}_{j,e-1}$ (UTXOs) can be discarded safely. We keep the regular blocks of epoch e, however, until after the end of epoch $e+1$ for the purpose of creating transaction proofs.

As OMNILEDGER's state is split across multiple shards and as we store only the state blocks' headers in a shard's ledger, a client cannot prove the existence of a past transaction to another party by presenting an inclusion proof to the block where the transaction was committed. We work around this by moving the responsibility of storing transactions' proofs-of-existence to the clients of OMNILEDGER. During epoch $e+1$ clients can generate proofs-of-existence for transactions validated in epoch e using the normal block of epoch e and the state block. Such a proof for a given transaction tx contains the Merkle tree inclusion proof to the regular

block B that committed tx in epoch e and a sequence of block headers from the state block $\mathsf{sb}_{j,e}$ at the end of the epoch to block B. To reduce the size of these proofs, state blocks can include several multi-hop backpointers to headers of intermediate (regular) blocks similarly to skipchains [195].

Finally, if we naively implement the creation of state blocks, it stalls the epoch's start, hence the transaction processing until $\mathsf{sb}_{j,e}$ has been appended to the ledger. To avoid this downtime, the consistent validators of the shard in epoch $e+1$ include an empty state-block at the beginning of the epoch as a place-holder; and once $\mathsf{sb}_{j,e}$ is ready they commit it as a regular block, pointing back to the place-holder and $\mathsf{sb}_{j,e-1}$.

6.5.4 Optional Trust-but-Verify Validation

There exists an inherent trade-off between the number of shards (and consequently the size of a shard), throughput and latency, as illustrated in Figure 6.4. A higher number of smaller shards results in a better performance but provides less resiliency against a more powerful attacker (25%). Because the design of OMNILEDGER favors security over scalability, we pessimistically assume an adversary who controls 25% of the validators and, accordingly, choose large shards at the cost of higher latency but guarantee the finality of transactions. This assumption, however, might not appropriately reflect the priorities of clients with frequent, latency-sensitive but low-value transactions (*e.g.,* checking out at a grocery store, buying gas or paying for coffee) and who would like to have transactions processed as quickly as possible.

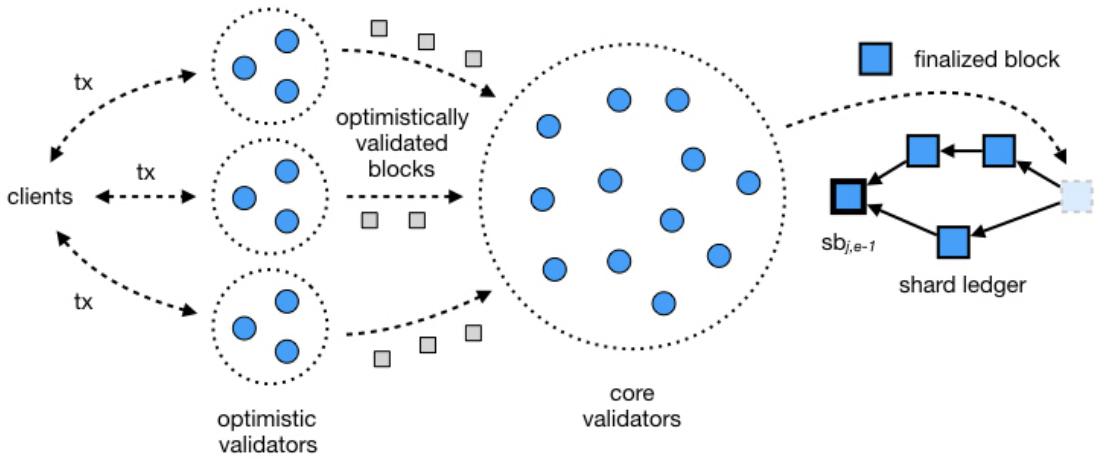

Figure 6.4 – Trust-but-Verify Validation Architecture

In response to the clients' needs, we augment the intra-shard architecture (see Figure 6.4) by following a *trust but verify* model, where *optimistic validators* process transactions quickly, providing a provisional but unlikely-to-change commitment and *core validators* subsequently verify again the transactions to provide finality and ensure verifiability. Optimistic validators follow the usual procedures for deciding which transactions are committed in which order; but they form much smaller groups, even as small as one validator per group. Consequently, they

produce smaller blocks with real-time latencies but are potentially less secure as the adversary needs to control a (proportionally) smaller number of validators to subvert their operation. As a result, some bad transactions might be committed, but ultimately core validators verify all provisional commitments, detecting any inconsistencies and their culprits, which makes it possible to punish rogue validators and to compensate the defrauded customers for the damages. The trust-but-verify approach strikes a balance for processing small transactions in real-time, as validators are unlikely to misbehave for small amounts of money.

At the beginning of an epoch *e*, all validators assign themselves to shards by using the per-epoch randomness and then bootstrap their states from the respective shard's last state block. Then, OMNILEDGER assigns each validator randomly to one of the multiple optimistic processing groups or a single core processing group. The shard-policy file specifies the number of optimistic and core validators, as well as the number of optimistic groups. Finally, in order to guarantee that any misbehavior will be contained inside the shard, it can also define the maximum amount of optimistic validated transactions to be equal to the stake or revenue of the validators.

Transactions are first processed by an optimistic group that produces optimistically validated blocks. These blocks serve as input for re-validation by core validators who run concurrently and combine inputs from multiple optimistic processing groups, thus maximizing the system's throughput (Figure 6.4). Valid transactions are included in a finalized block that is added to the shard's ledger and are finally included in the state block. However, when core validators detect an inconsistency, then the respective optimistically validated transaction is excluded and the validators who signed the invalid block are identified and held accountable, *e.g.,* by withholding any rewards or by excluding them from the system. We remark that the exact details of such punishments depend on the incentive scheme that is out of the scope of this work. Given a minimal incentive to misbehave and the quantifiable confidence in the security of optimistic validation (Figure 6.5), clients can choose, depending on their needs, to take advantage of real-time processing with an optimistic assurance of finality or to wait to have their transaction finalized.

6.6 Security Analysis

Our contributions are mainly pragmatic rather than theoretical and in this section, we provide an informal security analysis supplementing the arguments in Sections 6.4 and 6.5.

6.6.1 Randomness Creation

RandHound assumes an honest leader who is responsible for coordinating the protocol run and for making the produced randomness available to others. In OMNILEDGER, however, we cannot always guarantee that an honest leader will be selected. Although a dishonest leader cannot affect the unbiasability of the random output, he can choose to withhold the

randomness if it is not to his liking, thus forcing the protocol to restart. We economically disincentivize such behavior and bound the bias by the randomized leader-election process.

The leader-election process is unpredictable as the adversary is bound by the usual cryptographic hardness assumptions and is unaware of (a) the private keys of the honest validators and (b) the input string x to the VRF function. Also, OMNILEDGER's membership is unpredictable at the moment of private key selection and private keys are bound to identities. As a result, the adversary has at most $m = 1/4$ chance per round to control the elected leader as he controls at most 25% of all nodes. Each time an adversary-controlled leader is elected and runs RandHound the adversary can choose to accept the random output, and the sharding assignment produced by it, or to forfeit it and try again in hopes of a more favorable yet still random assignment. Consequently, the probability that an adversary controls n consecutive leaders is upper-bounded by $P[X \geq n] = \frac{1}{4^n} < 10^{-\lambda}$. For $\lambda = 6$, the adversary will control at most 10 consecutive RandHound runs. This is an upper bound, as we do not include the exclusion of the previous leader from the consecutive elections.

6.6.2 Shard-Size Security

We previously made the assumption that each shard is collectively honest. This assumption holds as long as each shard has less than $c = \lfloor \frac{n}{3} \rfloor$ malicious validators, because ByzCoinX requires $n = 3f + 1$ to provide BFT consensus.

The security of OMNILEDGER's validator assignment mechanism is modeled as a random sampling problem with two possible outcomes (honest or malicious). Assuming an infinite pool of potential validators, we can use the binomial distribution (Eq. 6.1). We can assume random sampling due to RandHound's unpredictability property that guarantees that each selection is completely random; this leads to the adversarial power of at most $m = 0.25$.

$$P\left[X \leq \lfloor \frac{n}{3} \rfloor\right] = \sum_{k=0}^{n} \binom{n}{k} m^k (1-m)^{n-k} \tag{6.1}$$

To calculate the failure rate of one shard, *i.e.,* the probability that a shard is controlled by an adversary, we use the cumulative distributions over the shard size n, where X is the random variable that represents the number of times we pick a malicious node. Figure 6.5 (right) illustrates the proposed shard size, based on the power of the adversary. In a similar fashion, we calculate the confidence a client can have that an optimistic validation group is honest (left).

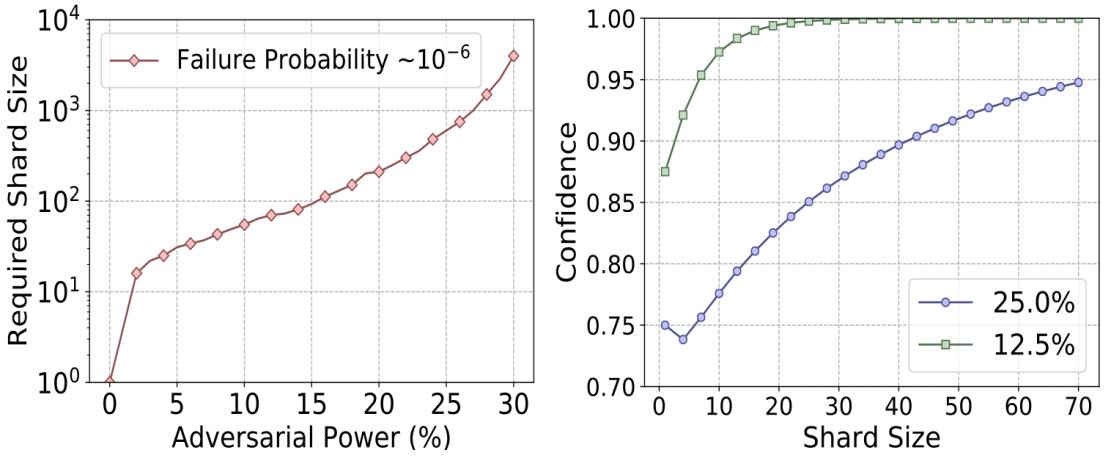

Figure 6.5 – Left: Shard size required for 10^{-6} system failure probability under different adversarial models. Right: Security of an optimistic validation group for 12.5% and 25% adversaries.

6.6.3 Epoch Security

In the last section, we modeled the security of a single shard as a random selection process that does, however, not correspond to the system's failure probability within one epoch. Instead, the total failure rate can be approximated by the union bound over the failure rates of individual shards.

We argue that, given an adequately large shard size, the epoch-failure probability is negligible. We can calculate an upper bound on the total-failure probability by permitting the adversary to run RandHound multiple times and select the output he prefers. This is a stronger assumption than what RandHound permits, as the adversary cannot go back to a previously computed output if he chose to re-run RandHound. An upper bound of the epoch failure event X_E is given by

$$P[X_E] \leq \sum_{k=0}^{l} \frac{1}{4^k} \cdot n \cdot P[X_S] \tag{6.2}$$

where l is the number of consecutive views the adversary controls, n is the number of shards and $P[X_S]$ is the failure probability of one shard as calculated in Section 6.6.2. For $l \to \infty$, we get $P[X_E] \leq \frac{4}{3} \cdot n \cdot P[X_S]$. More concretely, the failure probability, given a 12.5%-adversary and 16 shards, is $4 \cdot 10^{-5}$ or one failure in 68.5 years for one-day epochs

6.6.4 Group Communication

We now show that OMNILEDGER's group-communication pattern has a high probability of convergence under faults. We assume that there are N nodes that are split in $\sqrt{N}$ groups of $\sqrt{N}$ nodes each.

Setting the Time-Outs

In order to ensure that the shard leader will have enough time to find honest group leaders, we need to set up the view change time-outs accordingly. OMNILEDGER achieves this by having two time-outs. The first timeout T_1 is used by the shard leader to retry the request to non-responsive group members. The second timeout T_2 is used by the group members to identify a potential failure of a shard leader and to initiate a view-change [55]. To ensure that the shard leader has enough time to retry his requests, we have a fixed ratio of $T_1 = 0.1T_2$. However, if the T_2 is triggered, then in the new view T_2 doubles (as is typical [55]) in order to compensate for an increase in the network's asynchrony, hence T_1 should double to respect the ratio.

Reaching Consensus

We calculate the probability for a group size $N = 600$ where $\sqrt{N} = 25$: Given a population of 600 nodes and a sampling size of 25, we use the hypergeometric distribution for our calculation which yields a probability of 99.93% that a given group will have less than $25-10 = 15$ malicious nodes. A union bound over 25 groups yields a probability of 98.25% that no group will have more than 15 malicious nodes. In the worst case, where there are exactly $\frac{1}{3}$ malicious nodes in total, we need all of the honest validators to reply. For a group that contains exactly 15 malicious nodes, the shard's leader will find an honest group leader (for ByzCoinX) after 10 tries with a probability of $1 - ((15/24)^{10}) = 98.6\%$. As a result, the total probability of failure is $1 - 0.986 * 0.9825 = 0.031$.

This failure does not constitute a compromise of the security of OMNILEDGER. Rather, it represents the probability of a failure for the shard leader who is in charge of coordinating the shard's operation. If a shard leader indeed fails, then a new shard leader will be elected having 97% probability of successfully reaching consensus.

6.6.5 Censorship Resistance Protocol

One issue existing in prior work [149, 173] that OMNILEDGER partially addresses is when a malicious shard leader censors transactions. This attack can be undetectable from the rest of the shard's validators. A leader who does not propose a transaction is acceptable as far as the state is concerned, but this attack can compromise the fairness of the system or be used as a coercion tool.

For this reason, we enable the validators to request transactions to be committed, because they think the transactions are censored. They can either collect those transactions via the normal gossiping process or receive a request directly from a client. This protocol can be run periodically (*e.g.*, once every 10 blocks). We denote $N = 3f + 1$ validators exist where at most f are dishonest.

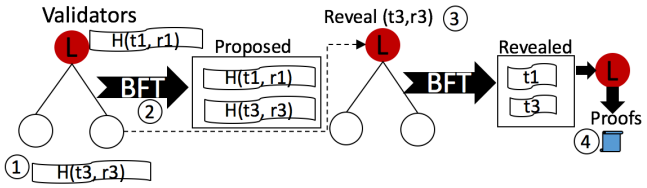

Figure 6.6 – Anti-censorship mechanism OMNILEDGER

The workflow (Figure 6.6), starts ① with each validator proposing a few (*e.g.*, 100)[3] blinded transactions for anti-censorship, which initiates a consensus round. The leader should add in the blocks all the proposals, however, he can censor f of the honest proposers. Nevertheless, he is blind on the f inputs he has to add from the honest validators he will reach consensus with. Once the round ends, there is a list ② of transactions that are eligible for anti-censorship, which is a subset of the proposed. As the transactions are blinded, no other validator knows which ones are proposed before the end of the consensus. Each validators reveals ③ his chosen transactions, the validators check that the transactions are valid and run consensus on which ones they expect the leader to propose. The leader is then obliged to include ④ the transactions that are consistent with the state, otherwise the honest validators will cause a view-change [55]. This way, if a validator sees transactions with normal fee-size being starved, he can preserve fairness.

6.7 Implementation

We implemented OMNILEDGER and its subprotocols for sharding, consensus, and processing of cross-shard transactions in Go [118]. For sharding, we combined RandHound's code, available on GitHub, with our implementation of a VRF-based leader-election mechanism by using a VRF construction similar to the one of Franklin and Zhang [107]. Similarly, to implement ByzCoinX, we extended ByzCoin's code, available on GitHub as well, by the parallel block commitment mechanism as introduced in Section 6.5.2. We also implemented the Atomix protocol, see Section 6.4.3, on top of the shards and a client that dispatches and verifies cross-shard transactions.

[3]The number should be small otherwise this protocol will take over the normal validation.

Table 6.1 – OMNILEDGER transaction confirmation latency in seconds for different configurations with respect to the shard size s, adversarial power f/n, and validation types.

$[s, f/n]$	[4, 1%]	[25, 5%]	[70, 12.5%]	[600, 25%]
Regular val.	1.38	5.99	8.04	14.52
1st lvl. val.	1.38	1.38	1.38	4.48
2nd lvl. val.	1.38	55.89	41.84	62.96

6.8 Evaluation

In this section, we experimentally evaluate our prototype implementation of OMNILEDGER. The primary questions we want to evaluate concern the overall performance of OMNILEDGER and whether it truly scales out (Section 6.8.2), the cost of epoch transitions (Section 6.8.3), the client-perceived latency when committing cross-shard transactions (Section 6.8.4), and the performance differences between ByzCoinX and ByzCoin with respect to throughput and latency (Section 6.8.5).

6.8.1 Experimental Setup

We ran all our experiments on DeterLab using 60 physical machines, each equipped with an Intel E5-2420 v2 CPU, 24 GB of RAM, and a 10 Gbps network link. To simulate a realistic, globally distributed deployment, we restricted the bandwidth of all connections between nodes to 20 Mbps and impose a latency of 100 ms on all communication links. The basis for our experiments was a data set consisting of the first 10,000 blocks of the Bitcoin blockchain.

6.8.2 OMNILEDGER Performance

In this experiment, we evaluate the performance of OMNILEDGER in terms of throughput and latency in different situations: we distinguish the cases where we have a fixed shard size and varying adversarial power (in particular 1%, 5%, 12.5%, and 25%) or the other way round. We also distinguish between configurations with regular or trust-but-verify validations where we use 1 MB blocks in the former case and 500 KB for optimistically validated blocks and 16 MB for final blocks in the latter case. In order to provide enough transactions for the final blocks, for each shard, there are 32 optimistic validation groups concurrently running; they all feed to one core shard, enabling low latency for low-risk transactions (Table 6.1) and high throughput of the total system.

Figure 6.7 shows OMNILEDGER's throughput for 1800 hosts in different configurations and, for comparison, includes the average throughput of Visa at ≈ 4000 tx/sec. Additionally, Table 6.1 shows the confirmation latency in the above configuration.

We observe that OMNILEDGER's throughput with trust-but-verify validation is almost an order

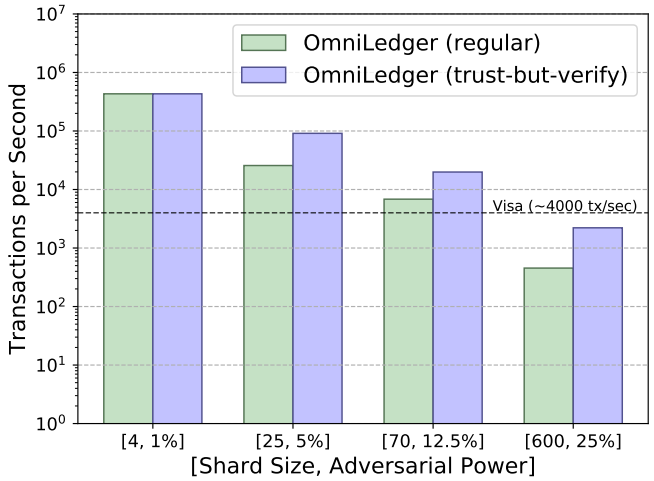

Figure 6.7 – OMNILEDGER throughput for 1800 hosts, varying shard sizes s, and adversarial power f/n.

Table 6.2 – OMNILEDGER scale-out throughput in transactions per second (tps) for a adversarial power of $f/n = 12.5\%$ shard size of $s = 70$, and a varying number of shards m.

m	1	2	4	8	16
tps	439	869	1674	3240	5850

of magnitude higher than with regular validation, at the cost of a higher latency for high-risk transactions that require both validation steps. For low-risk transactions, OMNILEDGER provides an optimistic confirmation in a few seconds after the first validation step, with less than 10% probability that the confirmation was vulnerable to a double-spending attack due to a higher-than-average number of malicious validators. For high-risk transactions, the latency to guarantee finality is still less than one minute.

Table 6.2 shows the scale-out throughput of OMNILEDGER with a 12.5% adversary, a shard size of 70, and a number of shards m between 1 and 16. As we can see, the throughput increases almost linearly in the number of shards.

In Figure 6.7, with a 12.5% adversary and a total number of 1800 hosts, we distributed the latter across 25 shards for which we measured throughput of 13,000 tps corresponding to 3 times the level of Visa. If we want to maintain OMNILEDGER's security against a 25% adversary and still achieve the same average throughput of Visa, *i.e.*, 4000 tps, then we estimate that we need to increase the number of hosts to about 4200 (which is less than the number of Bitcoin full nodes [32]) and split them into 7 shards. Unfortunately, our experimental platform could not handle such a high load, therefore, we mention here only an estimated value.

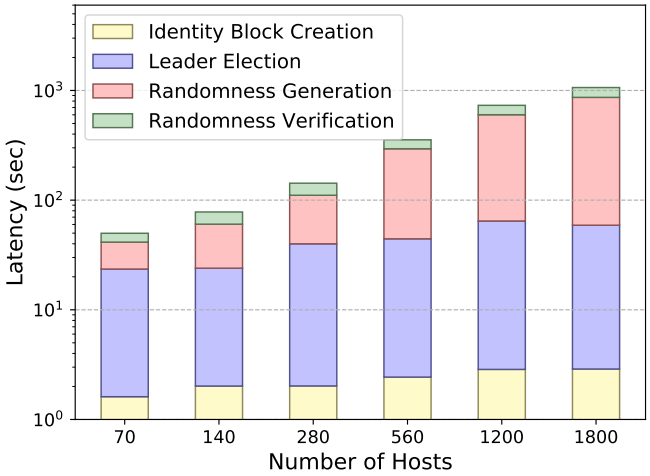

Figure 6.8 – Epoch transition latency.

6.8.3 Epoch-Transition Costs

In this experiment, we evaluate the costs for transitioning from an epoch $e-1$ to epoch e. Recall, that at the end of epoch $e-1$ the new membership configuration is first collectively signed, then used for the VRF-based leader-election. Once the leader is elected, he runs RandHound with a group-size of 16 hosts (which is secure for a 25% adversary [242]) and broadcasts it to all validators, who then verify the result and connect to their peers. We assume that validators already know the state of the shard they will be validating. It is important to mention that this process is not on the critical path, but occurs concurrently with the previous epoch. Once the new groups have been setup, the new shard leaders enforce a view-change.

As we can see in Figure 6.8, the cost of bootstrapping is mainly due to RandHound that takes up more than 70% of the total run time. To estimate the worst-case scenario, we refer to our security analysis in Section 6.6.1 and see that, even in the case with 1800 hosts, an honest leader is elected after 10 RandHound runs with high probability, which takes approximately 3 hours. Given an epoch duration of one day, this worst-case overhead is acceptable.

6.8.4 Client-Perceived End-to-End Latency with Atomix

In this experiment we evaluate in different shard configurations, the client-perceived, end-to-end latency when using Atomix. As shown in Figure 6.9, the client-perceived latency is almost double the value of the consensus latency as there are already other blocks waiting to be processed in the common case. Consequently, the inclusion of the transaction in a block is delayed. This latency increases slightly further when multiple shards validate a transaction. The overall end-to-end latency would be even higher if a client had to wait for output shards to run consensus that, however, is not required.

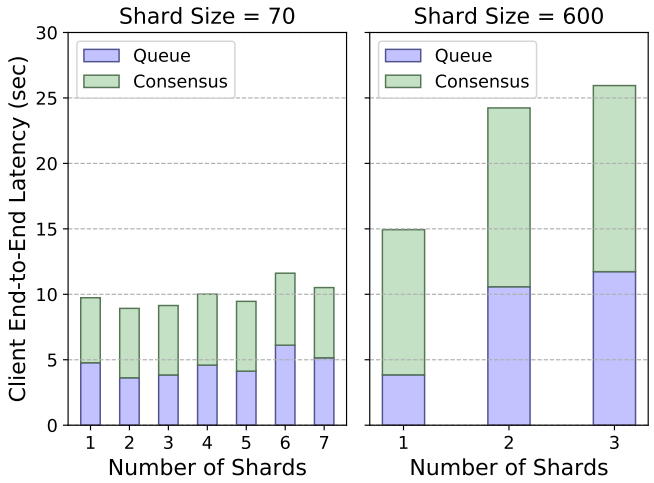

Figure 6.9 – Client-perceived, end-to-end latency for cross-shard transactions via Atomix.

Table 6.3 – ByzCoinX latency in seconds for different concurrency levels and data sizes.

	Concurrency			
Data Size	1	2	4	8
1 MB	15.4	13.5	12.6	11.4
8 MB	32.2	27.7	26.0	23.2
32 MB	61.6	58.0	50.1	50.9

If the client wants to spend the new funds directly, he can batch together the proof-of-acceptance and the expenditure transaction in order to respect the input-after-output constraint.

Overall, the client-perceived end-to-end latency for cross-shard transactions is not significantly affected when increasing the number of shards.

6.8.5 ByzCoinX Performance

In this experiment, we measure the performance improvements of ByzCoinX over the original ByzCoin. To have a fair comparison, each data-series corresponds to the total size of data concurrently in the network, meaning that if the concurrency level is 2 then there are 2 blocks of 4 MB concurrently, adding to a total of 8 MB, whereas a concurrency level of 4 means 4 blocks of 2 MB each.

In Figures 6.10 and Table 6.3, we see that there is a 20% performance increase when moving from one big block to four smaller concurrently running blocks, with a *concurrent* 35% decrease in the per-block consensus latency. This can be attributed to the higher resource utilization of the system, when blocks arrive more frequently for validation. When the concurrency further increases, we can see a slight drop in performance, meaning that the overhead

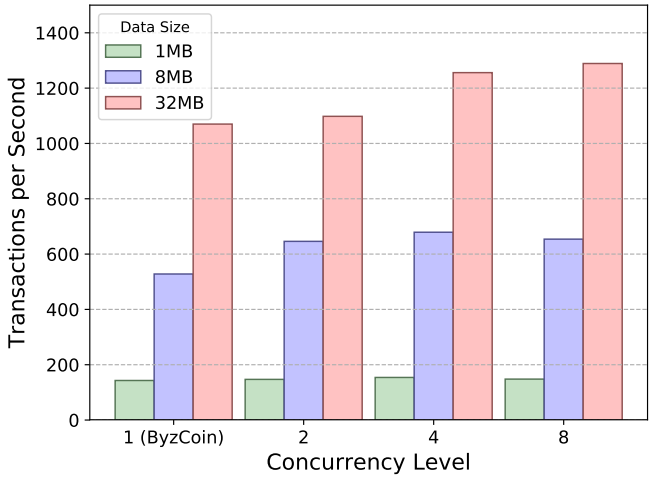

Figure 6.10 – ByzCoinX throughput in transactions per second for different levels of concurrency.

of the parallel consensus outweighs the parallelization benefits, due to the constant number of cryptographic operations per block.

Figure 6.11 illustrates the scalability of ByzCoin's [149] tree and fall-back flat topology, versus ByzCoinX's more fault-tolerant (group-based) topology and its performance when failures occur. As expected the tree topology scales better, but only after the consensus is run among more than 600 nodes, which assumes an adversary stronger than usual (see Figure 6.5).

For a group size below 600, ByzCoinX's communication pattern actually performs better than ByzCoin's. This is due to ByzCoinX's communication pattern that can be seen as a shallow tree where the roundtrip from root to leaves is faster than in the tree of ByzCoin. Hence, ByzCoin has a fixed branching factor and an increasing depth, whereas ByzCoinX has a fixed depth and an increasing branching factor. The effect of these two choices leads to better latencies for a few hundred nodes for fixed depth. The importance of the group topology, however, is that it is more fault tolerant because when failures occur the performance is not seriously affected. This is not true for ByzCoin; it switches to a flat topology in case of failure that does not scale after a few hundred nodes, due to the huge branching factor. This experiment was run with 1 MB blocks, the non-visible data point is at 300 seconds.

6.8.6 Bandwidth Costs for State Block Bootstrapping

In this experiment, we evaluate the improvements that state blocks offer to new validators during bootstrapping. Recall, that during an epoch transition, a new validator first crawls the identity blockchain, after which he needs to download only the latest state block instead of replaying the full blockchain to create the UTXO state. For this experiment, we reconstructed Bitcoin's blockchain [35, 219] and created a parallel OMNILEDGER blockchain with weekly

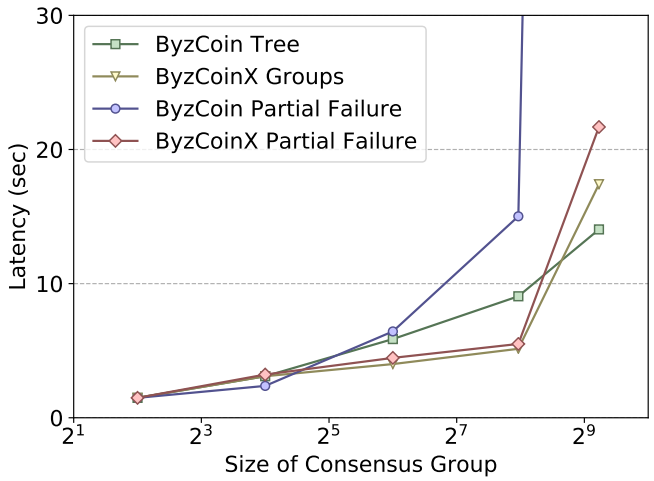

Figure 6.11 – ByzCoinX communication pattern latency.

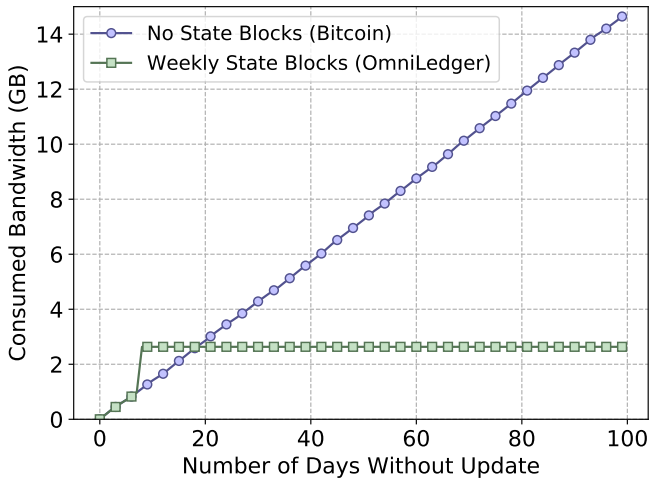

Figure 6.12 – Bootstrap bandwidth consumption with state blocks.

state blocks.

Figure 6.12 depicts the bandwidth overhead of a validator that did not follow the state for the first 100 days. As we can see, the state block approach is better if the validator is outdated for more than 19 days or 2736 Bitcoin blocks.

The benefit might not seem substantial for Bitcoin, but in OMNILEDGER, 2736 blocks are created in less than 8 hours, meaning that for one day-long epochs, the state block approach is significantly better. If a peak throughput is required and 16 MB blocks are deployed, we expect reduced bandwidth consumption close to two orders of magnitude.

6.9 Limitation and Future Work

OMNILEDGER is still a proof of concept and has limitations that we want to address in future work. First, even if the epoch bootstrap does not interfere with the normal operation, its cost (in the order of minutes) is significant. We leave to future work the use of advanced cryptography, such as BLS [39] for performance improvements. Additionally, the actual throughput is dependent on the workload as shown in Chainspace [6]. If all transactions touch all the shards before committing, then the system is better off with only one shard. We leave to future work the exploration of alternative ways of sharding, *e.g.*, using locality measures. Furthermore, we rely on the fact that honest validators will detect that transactions are unfairly censored and change the leader in the case of censorship. But, further anti-censorship guarantees are needed. We provide a protocol sketch in Section 6.6.5 and leave to future work its implementation and further combination with secret sharing techniques for providing stronger guarantees. Another shortcoming of OMNILEDGER is that it does not formally reason around the incentives of participants and focus on the usual honest or malicious model, which can be proven unrealistic in anonymous open cryptocurrencies. Finally, the system is not suitable for highly adaptive adversaries, as the bootstrap time of an epoch is substantial and scales only moderately, thus leading to the need for day-long epochs.

6.9.1 Atomix for State-full Objects

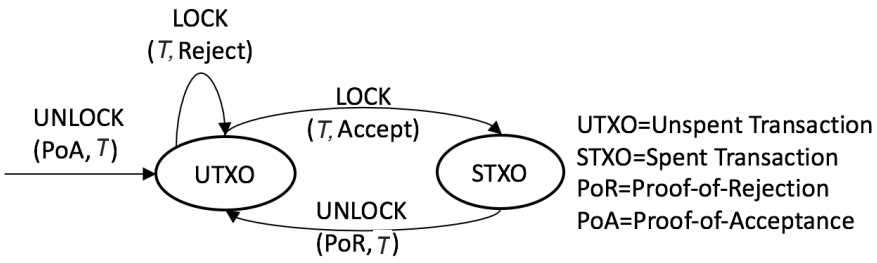

Figure 6.13 – State-Machine for the UTXO model. No locking is necessary

The original Atomix protocol in Section 6.4 implements a state machine as depicted in Fig-

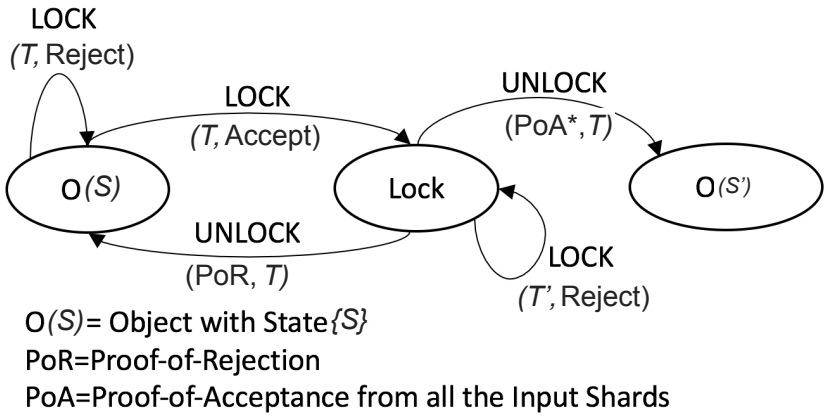

Figure 6.14 – State-Machine for the account model. Pessimistic locking is necessary

ure 6.13. We leave for future work the implementation of Atomix for state-full objects as described below.

To enable the use of such an algorithm in smart contracts we need to account on the fact that a smart-contract object is mutable and can be accessed concurrently for a legitimate reason. As a result, we need to modify the algorithm in two ways: a) the Unlock transactions should be sent to both Input and Output shards and b) the state machine should have one more state as the shards need to wait for confirmation before unlocking. This is necessary because there is the chance that the (state-full) object will be accessed again and this could violate the input-after-output dependency if Atomix decides to abort.

In Figure 6.14, we can see that an object will Lock for a specific transaction (T) and will reject any concurrent T', until T is committed and the new state S' is logged, or aborted and the old state S is open for change again.

6.10 Conclusion

OMNILEDGER is the first DL that securely scales-out to offer a Visa-level throughput and a latency of seconds while preserving full decentralization and protecting against a Byzantine adversary. OMNILEDGER achieves this through a novel approach consisting of three steps. First, OMNILEDGER is designed with concurrency in mind; both the full system (through sharding) and each shard separately (through ByzCoinX) validate transactions in parallel, maximizing resource utilization while preserving safety. Second, OMNILEDGER enables any user to transact

safely with any other user, regardless of the shard they use, by deploying *Atomix*, an algorithm for cross-shard transactions as well as real-time validation with the introduction of a trust-but-verify approach. Finally, OMNILEDGER enables validators to securely and efficiently switch between shards, without being bound to a single anti-Sybil attack method and without stalling between reconfiguration events.

We implemented and evaluated OMNILEDGER and each of its sub-components. ByzCoinX improves ByzCoin both in performance, with 20% more throughput and 35% less latency, and in robustness against failures. Atomix offers secure processing of cross-shard transactions and its overhead is minimal compared to intra-shard consensus. Finally, we evaluated the OMNILEDGER prototype thoroughly and showed that it can indeed achieve Visa-level throughput.

7 CALYPSO: Verifiable Management of Private Data over Blockchains

7.1 Introduction

New data privacy legislation, such as the European Union General Data Protection Regulation (GDPR) [94] or the EFF's call for information fiduciary rules for businesses [223], has reignited interest in secure management of private data. The ability to effectively share and manage data is one of the cornerstones of the digital revolution that turned many applications and processes to be data-driven and data-dependent. However, the current model of data management via a single server has repeatedly proven to be insecure and unfair: centralized data-sharing makes it possible to covertly grant access to unauthorized parties [164], and centralized data life-cycle management can fail to enforce all phases of the cycle and "forget" to delete user data [108]. Lastly, well-located servers may have an unfair advantage in accessing information faster than others, thereby, for example, requiring additional burdensome regulations for trading markets [152].

Replacing single points of failure with decentralized alternatives is an obvious approach to achieve more transparent and fair data sharing and management. Decentralized data-sharing can give rise to data markets controlled by users [237] and not solely by tech giants such as Google or Facebook; enable sharing of confidential data between mutually distrustful parties, such as state institutions or even different countries; or bring the much-needed transparency to lawful access requests [98]. Decentralized data life-cycle management can enable effective and guaranteed data retention (*e.g.*, legal or corporate data retention policies or enforcement of the "right to be forgotten"), or an implementation of an information-publication version of a dead man's switch [91] that enables journalists to create a contingency plan to have entrusted data disclosed under specific circumstances. Finally, if correctly implemented, decentralized data life-cycle management can also result in fair lotteries [12], games (*e.g.*, poker [157]), and trading (*e.g.*, exchanges [73]).

Unfortunately, current decentralized data-sharing applications [170, 203] fail to manage private data securely unless they forfeit the full life-cycle management [109] and publish encrypted data on Bitcoin; or rely on semi-centralized solutions [15, 256]. Furthermore, de-

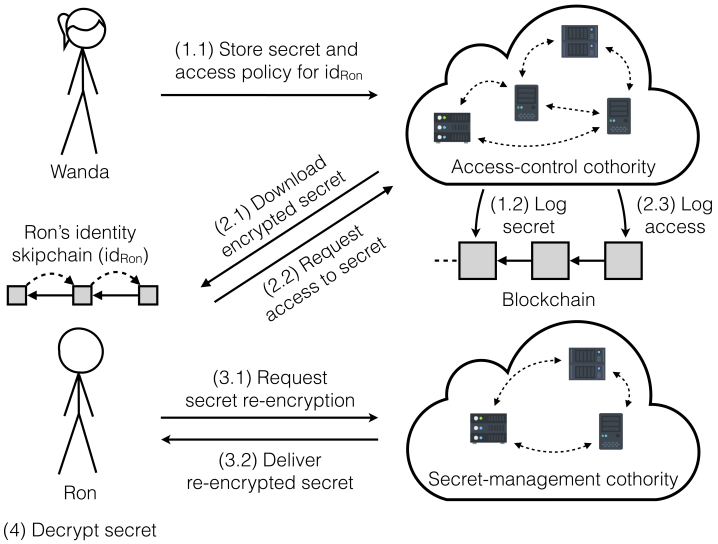

Figure 7.1 – Auditable data sharing in CALYPSO: (1) Wanda encrypts data under the secret-management cothority's key, specifying the intended reader (*e.g.,* Ron) and the access policy, and then sends it to the access-control cothority which verifies and logs it. (2) Ron downloads the encrypted secret from the blockchain and then requests access to it by contacting the access-control cothority which logs the query if valid, effectively authorizing Ron's access to the secret. (3) Ron asks the secret-management cothority for the secret shares of the key needed to decrypt the secret by proving that the previous authorization by access-control cothority was successful. (4) Ron decrypts the secret. If a specific application requires fairness, the data can be atomically disclosed on-chain.

centralized applications that rely on the timing of data disclosure to enforce fairness are susceptible to front-running attacks where the adversary gets early access to information and unfairly adapts their strategies. For example, the winner of the Fomo3D [225] event (gaining a prize of 10.5k Ether ($2.2M at the time)) enforced an early termination of the lottery by submitting a sequence of high-fee transactions, which significantly increased his winning probability. Due to the lack of fairness guarantees, decentralized exchanges remain vulnerable [73] or resort to centralized order-book matching (*e.g.,* 0x Project [64]), and decentralized lotteries require collateral [12] or run in a non-constant number of rounds [183].

In this paper, we introduce CALYPSO, a new secure data-management framework that addresses the challenge of providing fair and verifiable access to private information without relying on a trusted party. In achieving this goal CALYPSO faces three key challenges. First, CALYPSO has to provide accountability for all accesses to confidential data to ensure that secrets are not improperly disclosed and to enforce the proper recording of data accesses and usage. Second, CALYPSO has to prevent front-running attacks and guarantee fair access to information. Third, CALYPSO has to enable data owners to maintain control over the data they share, and data consumers to maintain access even when their identities (public keys) are updated. In particular, CALYPSO should allow for flexible updates to access-control rules

and user identities, *e.g.,* to add or revoke access rights or public keys. Figure 7.1 provides an overview of a typical data-sharing application using CALYPSO that builds on top of a novel abstraction called *on-chain secrets* (OCS) and uses skipchains [195, 148] to provide dynamic access-control and identity management.

On-chain secrets addresses the first two challenges by combining threshold cryptography [222, 227, 229] and blockchain technology [149, 250] to enable users to share their encrypted data with *collective authorities* (cothorities) that are responsible for enforcing access-control and atomically disclosing data to authorized parties. Furthermore, CALYPSO combines on-chain secrets with skipchains [148, 195] in order to enable dynamic access-control and identity-management. We present two specific secure implementations of on-chain secrets, namely *one-time secrets* and *long-term secrets*, that have different trade-offs in terms of computational and storage overheads.

To evaluate CALYPSO, we implemented a prototype in Go and ran experiments on commodity servers. We implemented both versions of on-chain secrets and show that they have a moderate overhead of 0.2 to 8 seconds for cothorities of 16 and 128 trustees, respectively, and overall scale linearly in the number of trustees. Furthermore, in addition to evaluations on simulated data, we tested two deployments of CALYPSO using real data traces. First, we deployed a document-sharing application and tested it under different loads. CALYPSO takes 10–20 (10–150) seconds to execute a write (read) request, and has a 0.2× to 5× latency overhead compared to a semi-centralized solution that stores data on the cloud. Finally, we show that CALYPSO-based zero-collateral lotteries significantly outperform the state-of-the-art as they require 1 and $\log n$ rounds to finish, respectively, where n denotes the number of lottery participants.

In summary, this paper makes the following contributions.

- We introduce CALYPSO, a decentralized framework for auditable management of private data while maintaining fairness and confidentiality (see Section 7.3). CALYPSO enables dynamic updates to access-control rules without compromising security.

- We present on-chain secrets and its two implementations, one-time and long-term secrets, that enable transparent and efficient management of data without requiring a trusted third party (see Section 7.4).

- We demonstrate the feasibility of using CALYPSO to address the data sharing needs of actual organizations by presenting three classes of realistic, decentralized deployments: auditable data sharing, data life-cycle management, and atomic data publication (see Section 7.7). To evaluate our system and conduct these feasibility studies, we created an implementation of CALYPSO which was independently audited and will be released as open-source (see Sections 7.8 and 7.9).

7.2 Motivating Applications

In this section, we motivate the practical need for CALYPSO by describing how it can enable security and fairness in three different classes of applications: auditable data sharing, data life-cycle management, and atomic data publication.

7.2.1 Auditable Data Sharing.

Current cloud-based data-sharing systems (*e.g.,* Dropbox or Google Drive) provide a convenient way to store and share data, however, their main focus is on integrity whereas ensuring data confidentiality and access accountability is often secondary if provided at all. Further, clients must trust the individual companies that these properties are indeed achieved in practice as clients, for a variety of reasons, are typically not able to verify these security guarantees themselves. Furthermore, many systems often provide only retroactive and network-dependent detection mechanisms for data integrity failures [168, 177]. Consequently, any secure data-sharing system should be decentralized to avoid the need to rely on trusted third parties, and it should provide integrity, confidentiality, and accountability and ensure that any violations can be detected proactively.

Current decentralized data-sharing applications [170, 203, 239, 251] that focus on shared access to private data often fail to manage these private data in a secure and accountable manner, especially when it comes to sharing data between independent and mutually-distrustful parties. They either ignore these issues altogether [139, 253], fall back on naive solutions [109], or use semi-centralized approaches [15, 138, 256], where access information and hashes of the data are put on-chain but the secret data is stored and managed off-chain, hence violating the accountability requirement.

To address the above challenges and enable secure decentralized data-sharing applications, CALYPSO uses threshold cryptography and distributed-ledger technology to protect the integrity and confidentiality of shared data and to ensure data-access accountability by generating a third-party verifiable audit trail for data accesses. Designers of decentralized applications can further use CALYPSO to achieve additional functionalities such as monetizing data accesses or providing proofs to aid investigations of data leaks or breaches.

7.2.2 Data Life-Cycle Management.

Custodian systems can provide secure data life-cycle management and have a variety of useful applications. However, certain features, such as provable deletion and provable publication, are problematic to achieve in practice. The lack of provable deletion mechanisms affects users who want to enforce their legal *right to be forgotten* provided by the EU GDPR legislature [94]. This situation is particularly tricky when the relevant data is stored in the databases of large organizations, such as Google or Facebook, whose systems were not designed to provide

this feature. Provable publication of data based on user-specified policies would permit automatic publication of documents, for example, legal wills or estate plans, but only when specific conditions are met. Further, it would enable whistleblowers to implement digital life insurances where files are published automatically unless the custodian receives a regular heartbeat message that bears the digital signature of the insured person [91, 215].

Securely realizing such custodian systems is a challenging task as the intuitive, centralized designs would not protect users from malicious, bribed or coerced providers. Custodian systems also need to provide support for (decentralized) access-control mechanisms to effectively express and enforce rules, which is necessary to enable applications such as the ones described above. Previous works have already attempted to design such systems but they have weak spots, *e.g.,* in terms of failure resilience or guaranteed erasure of data [108].

CALYPSO solves these challenges with on-chain secrets and an expressive access-control mechanism that enables the revocation of access rights for everyone, effectively preventing any access to the secrets stored on-chain. Furthermore, the access-control mechanism can express not only time-based but also event-based public decryption (*e.g.,* reveal data in the absence of a heartbeat message).

7.2.3 Atomic Data Publication.

Security and fairness requirements significantly change when an application is deployed in a Byzantine, decentralized environment as opposed to the traditional, centralized setting. For example, an adversary can easily gain an unfair advantage over honest participants through front-running attacks [73, 225, 241] if decentralized applications, such as lotteries [12], poker games [157], or exchanges [73], are not designed with such attacks in mind. To protect users against front-running, these applications often fall back to trusted intermediaries giving up decentralization, or they implement naive commit-and-reveal schemes exposing themselves to liveness attacks where adversaries can DoS the application or force it to restart by refusing to reveal their inputs. To provide failure resilience and protect against such late aborts, many applications introduce complex incentive mechanisms whose security guarantees are usually difficult to analyze [12, 157].

In CALYPSO, all inputs committed by the participants (*e.g.,* lottery randomness, trading bids, game moves) remain confidential up to a barrier point that is expressed through specific rules defined in a policy. All of the decommitted values are taken into account to compute and atomically disclose the results of the protocol to every interested party. Consequently, CALYPSO resolves the tension between decentralization, fairness, and availability, and provides a secure foundation for decentralized applications.

7.3 CALYPSO Overview

This section provides an overview of CALYPSO. We start with a strawman solution to motivate the challenges that any secure, decentralized data-sharing and management system should address. We then derive the system goals from our observations, describe system and threat models, and lastly present the system design (see Figure 7.1).

7.3.1 Strawman Data Management Solution

We assume that the strawman consists of a tamper-resistant public log, such as the Bitcoin blockchain, and that participants register their identities on-chain, *e.g.,* as PGP keys. Now consider an application on top of the strawman system where Wanda is the operator of a paid service providing asynchronous access to some information and Ron is a customer. Once Ron has paid the fee, Wanda can simply encrypt the data under Ron's key, post the ciphertext on-chain which Ron can then retrieve and decrypt at his discretion.

This strawman approach provides the intended functionality but has several drawbacks. (1) There is no data access auditability because Ron's payment record does not prove that he actually accessed the data, and such a proof might be needed if the provided information is misused, for example. (2) If Wanda ever wants to change the access rights, *e.g.,* because Ron cancelled his access subscription, she cannot do so because the ciphertext is on-chain and Ron controls the decryption key. (3) If Ron ever needs to change his public key, he would lose access to all data encrypted under that key, unless Wanda re-publishes that data using Ron's new key. (4) Since the exchange of payment and data is not atomic, having submitted a payment successfully does not guarantee that Ron access to the data. (5) If Wanda makes the data available on a first-come-first-serve basis, then customers with better connectivity are able to make payments faster and thereby mount front-running attacks.

To address these issues, we introduce two new components and transform the strawman into CALYPSO.

1. To enable auditability of data accesses and ensure atomic data delivery, we introduce *on-chain secrets* (OCS) in Sections 7.4.1 and 7.4.2.

2. To enable decentralized, dynamic, user-sovereign identities and access policies, we extend skipchains and integrate them with CALYPSO in Section 7.5.

7.3.2 System Goals

CALYPSO has the following primary goals.

- **Confidentiality:** Secrets stored on-chain can only be decrypted by authorized clients.

- **Auditability:** All access transactions are third-party verifiable and recorded in a tamper-resistant log.
- **Fair access:** Clients are guaranteed to get access on a secret they are authorized for if any only if they posted an access request on-chain. If a barrier point exists, authorized clients get concurrent access after it.
- **Dynamic sovereign identities:** Users (or organizations) fully control their identities (public keys) and can update them in a third-party verifiable way.
- **Decentralization:** No single point of compromise or failure.

7.3.3 System Model

There are four main entities in CALYPSO: *writers* who put secrets on-chain, *readers* who retrieve secrets, an *access-control collective authority* that is responsible for logging write and read transactions on-chain and enforcing access control for secrets, and a *secret-management collective authority* that is responsible for managing and delivering secrets. In the rest of the paper, we use Wanda and Ron to refer to a (generic) writer and reader, respectively.

A collective authority or cothority is an abstract decentralized entity that is responsible for some authoritative action. We call the nodes of a cothority *trustees*. For example, the set of Bitcoin miners can be considered a cothority that maintain the consistency of Bitcoin's state. The access-control cothority requires a Byzantine fault-tolerant consensus [149, 150, 159, 191]. There are various ways to implement an access-control cothority, *e.g.,* as a set of permissioned servers that maintains a blockchain using BFT consensus or as an access-control enforcing a smart contract on top of a permissionless cryptocurrency such as Ethereum. The secret-management cothority membership is fixed; it may be set up on a per-secret basis or in a more persistent setting, the differences of which are discussed in Section 7.4. The secret-management trustees maintain their private keys and may need to maintain additional secret state, such as private-key shares. They do not run consensus for every transaction.

We denote private and public key pairs of Wanda and Ron by $(\mathsf{sk}_W, \mathsf{pk}_W)$ and $(\mathsf{sk}_R, \mathsf{pk}_R)$. Analogously, we write $(\mathsf{sk}_i, \mathsf{pk}_i)$ to refer to the key pair of trustee i. To denote a list of elements we use angle brackets, *e.g.,* we write $\langle \mathsf{pk}_i \rangle$ to refer to a list of public keys $\mathsf{pk}_1, \ldots, \mathsf{pk}_n$. We assume that there is a registration mechanism through which writers have to register their public keys pk_W on the blockchain before they can start any secret-sharing processes. We denote an access-control label by policy, where policy = pk_R is the simplest case with Ron being the only reader.

7.3.4 Threat Model

We make the usual cryptographic assumptions: the adversary is computationally bounded, cryptographically-secure hash functions exist, and there is a cyclic group $\mathscr{G}$ (with generator

g) in which the decisional Diffie-Hellman assumption holds. We assume that participants, including trustees, verify the signatures of the messages they receive and process those that are correctly signed.

In the respective cothorities, we denote the total number of trustees by n and those that are malicious by f. Depending on the consensus mechanism that is used for the blockchain underlying the access-control cothority, we either require an honest majority $n = 2f + 1$ for Nakamoto-style consensus [191] or $n = 3f + 1$ for classic BFT consensus [149][1]. In the secret-management cothority, we require $n = 2f + 1$ and set the threshold to recover a secret to $t = f + 1$.

We assume that readers and writers do not trust each other. We further assume that writers encrypt the correct data and share the correct symmetric key with the secret-management cothority, as readers can release a protocol transcript and prove the misbehavior of writers. Conversely, readers might try to get access to a secret and claim later that they have never received it. Additionally, writers might try to frame readers by claiming that they shared a secret although they have never done so. Finally, the writer can define a *barrier point*, an event before which no one can access the secret guaranteeing fair access. We guarantee data confidentiality up to the point where an authorized reader gains access. To maintain confidentiality after this point, writers may rely on additional privacy-preserving technologies such as differential privacy [83] or homomorphic encryption [97].

7.3.5 Architecture Overview

On a high level CALYPSO enables Wanda, the writer, to share a secret with Ron, the reader, under a specific access-control policy. When Wanda wants to put a secret on-chain (see Figure 7.1), she encrypts the secret and then sends a write transaction tx_w to the access-control cothority. The access-control cothority verifies and logs tx_w, making the secret available for retrieval by Ron, the authorized reader. To request access to a secret, Ron downloads the secret from the blockchain and sends to the access-control cothority a read transaction tx_r which carries a valid authorization from Ron's identity skipchain with respect to the current policy.

If Ron is authorized to access the requested secret, the access-control cothority logs tx_r. Subsequently, Ron contacts the secret-management cothority to recover the secret. The secret-management trustees verify Ron's request using the blockchain and check that the barrier point (if any) has occurred. Afterwards, the trustees deliver the secret shares of the key needed to decrypt Wanda's secret as shared in tx_w.

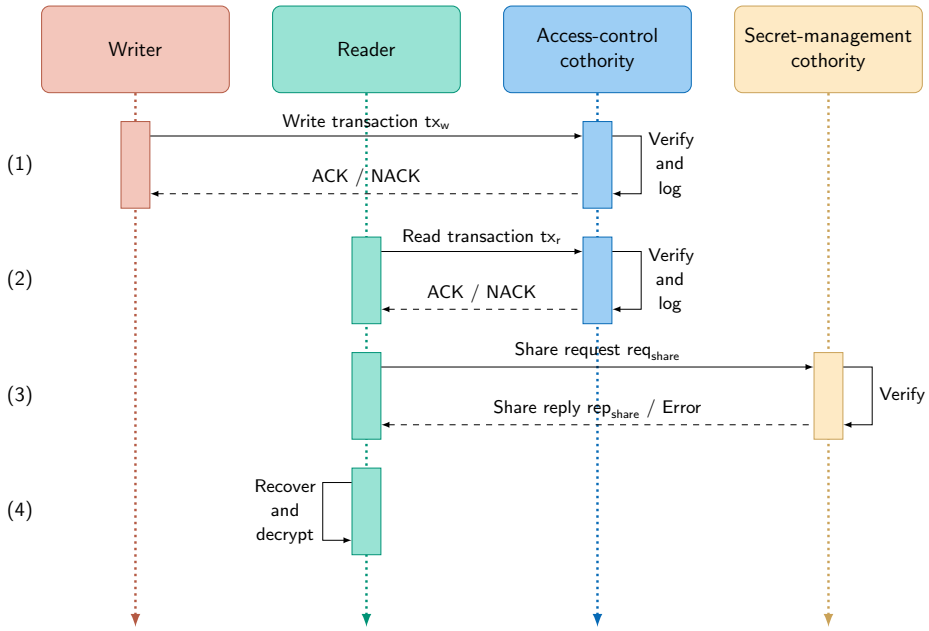

Figure 7.2 – On-chain secrets protocol steps: (1) Write transaction, (2) Read transaction, (3) Share retrieval, (4) Secret reconstruction.

7.4 On-Chain Secrets

In this section we introduce two on-chain secrets protocols, *one-time secrets* and *long-term secrets*. Figure 7.2 provides an overview of on-chain secrets. The assumptions listed in the previous section apply to both protocols. We also present two protocol extensions: an on-chain blinded key exchange that allows concealing the identities of the readers as well as a post-quantum secure version of on-chain secrets.

One-time secrets uses PVSS and employs a per-secret secret-management cothority. One-time secrets' simplicity enables each tx_w to define a fresh, ad hoc group of secret-management trustees without any setup or coordination. This simplicity, however, comes at a price. The tx_w size and the encryption/decryption overhead are linear in the size of the secret-management cothority because all encrypted shares are included in the transaction and the number of secret shares is equal to the size of the secret-management cothority.

Long-term secrets, the second approach to implementing on-chain secrets, requires the secret-management cothority to perform a coordinated setup phase to generate a collective key (DKG) and to maintain a minimal state of their DKG secret shares. As a result, however, long-term secrets offers a constant encryption overhead and a flexible secret-management cothority membership through re-sharing the shared key or re-encrypting existing secrets to a new shared key.

[1] We assume the associated network model is strong enough to guarantee the security of the blockchain used.

The *on-chain private key exchange protocol* can be applied to both on-chain secrets protocols and hide the reader's identity by blinding the reader's public key in the write and read transactions. Lastly, the *post-quantum on-chain secrets* describe the modifications needed to achieve post-quantum security.

7.4.1 One-Time Secrets

One-time secrets is based on PVSS [222]. Wanda, the writer, first prepares a secret she wants to share along with a policy that lists the public key of the intended reader. She then generates a symmetric encryption key by running PVSS for the secret-management cothority members, encrypts the secret with the key she shared and then stores the resulting ciphertext either on-chain or off-chain. Finally, she sends a write transaction tx_w containing the information necessary for the verification and retrieval of her secret to the access-control cothority to have it logged. Ron, the reader, creates and sends to the access-control cothority a read transaction tx_r for a specific secret. The trustees check tx_r against the secret's access policy and if Ron is authorized to access the secret, they log the transaction creating a proof of access. Ron sends this proof together with the encrypted secret shares from tx_w to each secret-management (PVSS) trustee and gets the secret key shares. Once Ron has received a threshold of valid shares, he recovers the symmetric key and decrypts the original data.

Write transaction protocol

Wanda, the writer and each trustee of the access-control cothority perform the following protocol to log the write transaction tx_w on the blockchain. Wanda initiates the protocol as follows.

1. Compute $h = \mathsf{H}(\text{policy})$ to map the access-control policy to a group element h to be used as the base point for the PVSS polynomial commitments.
2. Choose a secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ of degree $t-1$. The secret to be shared is $s = G^{s(0)}$.
3. For each secret-management trustee i, compute the encrypted share $\widehat{s_i} = \mathsf{pk}_i^{s(i)}$ of the secret s and create the corresponding NIZK proof $\pi_{\widehat{s_i}}$ that each share is correctly encrypted (see Section 2.3.6). Create the polynomial commitments $b_j = h^{a_j}$, for $0 \leq j \leq t-1$.
4. Set $k = \mathsf{H}(s)$ as the symmetric key, encrypt the secret message m to be shared as $c = \text{enc}_k(m)$, and compute $H_c = \mathsf{H}(c)$. Set $\text{policy} = \mathsf{pk}_R$ to designate Ron as the intended reader of the secret message m.
5. Finally, prepare and sign the write transaction

$$\text{tx}_\text{w} = [\langle \widehat{s_i} \rangle, \langle b_j \rangle, \langle \pi_{\widehat{s_i}} \rangle, H_c, \langle \mathsf{pk}_i \rangle, \text{policy}]_{\text{sig}_{\mathsf{sk}_W}}$$

and send it to the access-control cothority.

The access-control cothority then logs the write transaction on the blockchain as follows.

1. Derive the PVSS base point $h = \mathsf{H}(\text{policy})$.
2. Verify each encrypted share $\widehat{s_i}$ against $\pi_{\widehat{s_i}}$ using $\langle b_j \rangle$ and h (see Section 2.3.6). This step guarantees that Wanda correctly shared the encryption key.
3. If all shares are valid, log $\mathrm{tx_w}$ in block b_w.

Read transaction protocol

After the write transaction has been recorded, Ron needs to log the read transaction $\mathrm{tx_r}$ through the access-control cothority before he can request the secret. To do so, Ron performs the following steps.

1. Retrieve the ciphertext c and b_w, which stores $\mathrm{tx_w}$, from the access-control cothority.
2. Check that $\mathsf{H}(c)$ is equal to H_c in $\mathrm{tx_w}$ to ensure that the ciphertext c of Wanda's secret has not been altered.
3. Compute $H_\mathrm{w} = \mathsf{H}(\mathrm{tx_w})$ as the unique identifier for the secret that Ron requests access to and determine the proof $\pi_{\mathrm{tx_w}}$ showing that $\mathrm{tx_w}$ has been logged on-chain.
4. Prepare and sign the transaction

$$\mathrm{tx_r} = [H_\mathrm{w}, \pi_{\mathrm{tx_w}}]_{\mathrm{sig}_{\mathsf{sk}_R}}$$

and send it to the access-control cothority.

The access-control cothority then logs the read transaction on the blockchain as follows.

1. Retrieve $\mathrm{tx_w}$ using H_w and use pk_R, as recorded in policy, to verify the signature on $\mathrm{tx_r}$.
2. If the signature is valid and Ron is authorized to access the secret, log $\mathrm{tx_r}$ in block b_r.

Share retrieval protocol

After the read transaction has been logged, Ron can recover the secret message m by running first the share retrieval protocol with the secret-management cothority to obtain shares of the encryption key used to secure m. To do so, Ron initiates the protocol as follows.

1. Create and sign a secret-sharing request

$$\text{req}_{\text{share}} = [\text{tx}_\text{w}, \text{tx}_\text{r}, \pi_{\text{tx}_\text{r}}]_{\text{sig}_{\text{sk}_R}}$$

   where π_{tx_r} proves that tx_r has been logged on-chain.

2. Send $\text{req}_{\text{share}}$ to each secret-management trustee to obtain the decrypted shares.

Each trustee i of the secret-management cothority responds to Ron's request as follows.

1. Use pk_R in tx_w to verify the signature of $\text{req}_{\text{share}}$ and π_{tx_r} to check that tx_r has been logged on-chain.

2. Compute the decrypted share $s_i = (\widehat{s_i})^{\text{sk}_i^{-1}}$, create a NIZK proof π_{s_i} that the share was decrypted correctly (see Section 2.3.6), and derive $c_i = \text{enc}_{\text{pk}_R}(s_i)$ to ensure that only Ron can access it.

3. Create and sign the secret-sharing reply

$$\text{rep}_{\text{share}} = [c_i, \pi_{s_i}]_{\text{sig}_{\text{sk}_i}}$$

   and send it back to Ron.

Secret reconstruction protocol

To recover the secret key k and decrypt the secret m, Ron performs the following steps.

1. Decrypt each $s_i = \text{dec}_{\text{pk}_R}(c_i)$ and verify it against π_{s_i}.

2. If there are at least t valid shares, use Lagrange interpolation to recover s.

3. Recover the encryption key as $k = \text{H}(s)$ and use it to decrypt the ciphertext c to obtain the message $m = \text{dec}_k(c)$.

Achieving system goals

The one-time secrets protocol achieves all our goals except for dynamic sovereign identities.

Confidentiality of Secrets. The secret message m is encrypted under a symmetric key k which is securely secret-shared using PVSS among the secret-management trustees such that $t = f + 1$ shares are required to reconstruct it. The access-control trustees verify and log on the blockchain the encrypted secret shares which, based on the properties of PVSS, do not leak any information about k. After the secret-management trustees receive a valid request $\text{req}_{\text{share}}$, they respond with their secret shares encrypted under the public key listed in policy from

the respective write transaction tx_w. Further, a dishonest reader cannot obtain access to someone else's secret through a new write transaction that uses a policy that lists him as the reader but copies secret shares from another transaction in hopes of having them decrypted by the secret-management cothority. This is because each transaction is bound to a specific policy which is used to derive the base point for the PVSS NIZK consistency proofs. Without the knowledge of the decrypted secret shares (and the key k), the malicious reader cannot generate correct proofs and all transactions without valid proofs are rejected. This means that only the intended reader obtains a threshold of secret shares necessary to recover k and then access m.

Auditability. Under the assumption that the access-control cothority provides Byzantine consensus guarantees, all properly created read and write transactions are logged by the access-control cothority on the blockchain. Once a transaction is logged, anyone can verify this fact and obtain a third-party verifiable transaction inclusion proof.

Atomic Data Delivery. Once a read transaction tx_r is logged by the access-control cothority, the reader can run the share retrieval protocol with the secret-management cothority. Under the assumption that $n = 2f + 1$, the reader receives at least $t = f + 1$ shares of the symmetric encryption key k from the honest trustees. This guarantees that the reader has enough shares to reconstruct k and access the secret message m using the secret reconstruction protocol.

Dynamic Sovereign Identities. While all participants maintain their own private keys and hence their identities, the identities used in write transactions cannot be updated without re-encrypting the secrets and posting new write transactions.

Decentralization. The protocols do not assume a trusted third party and they tolerate up to $t - 1$ failures.

Protocol advantages and shortcomings

The one-time secrets protocol uses existing and proven to be secure building blocks and its design is simple to implement and analyze. Further, it does not require a setup phase among the secret-management members, *e.g.*, to generate a collective private-public key pair. It also enables the use of a different secret-management cothority for each secret, without requiring the servers to maintain any protocol state.

However, one-time secrets has a few shortcomings too. First, it incurs high PVSS setup and share reconstruction costs as Wanda needs to evaluate the secret sharing polynomial at n points, create n encrypted shares and NIZK proofs, along with t polynomial commitments. Similarly, Ron has to verify up to n decrypted shares against the NIZK proofs and to reconstruct the secret on his device. Second, the transaction size increases linearly with the secret-management cothority size. Because the secret-management trustees do not store any per-secret protocol state making them nearly stateless, the write transaction tx_w must contain the encrypted shares, NIZK proofs, and the polynomial commitments. Lastly, one-time se-

crets does not enable re-encryption of the shares to another set of trustees, preventing the possibility of moving shares from one set of secret-management trustees to another.

7.4.2 Long-Term Secrets

Long-term secrets address the above limitations through a dedicated secret-management cothority that persists over a long period of time and that maintains a collective private-public key pair used to secure access to the secrets.

After a one-time distributed key generation (DKG) phase (see Section 2.3.1 for details) performed by the secret-management cothority, Wanda, the writer, prepares her secret message, encrypts it with a symmetric key and then encrypts that key with the shared public key of the secret-management cothority. As a result, the overhead of encrypting secrets is constant as each write transaction contains a single ciphertext instead of individual shares. Ron, the reader, recovers the symmetric key by obtaining a threshold of securely blinded shares of the collective private key and reconstructing the symmetric key himself or with the help of a trustee he selects. Furthermore, the configuration of the secret-management cothority can change by re-sharing the shared key or re-encrypting all the secrets to a new secret-management cothority.

Protocol setup

Before any transactions can be created and processed, the secret-management cothority needs to run a DKG protocol to generate a shared private-public key pair. There exist a number of DKG protocols that are synchronous [110] or asynchronous [141]. Given the rarity of the setup phase, we assume a pessimistic synchrony assumption for the DKG (*e.g.*, every message is posted to the access-control cothority blockchain) and implement the DKG by Gennaro et al. [110] because of its simplicity and the fact that it permits a higher threshold of corruptions.

The output of the setup phase is a collective public key $\mathsf{pk}_{\text{smc}} = G^{\mathsf{sk}_{\text{smc}}}$, where sk_{smc} is the unknown collective private key. Each server i holds a share of the secret key denoted as sk_i and all servers know the public counterpart $\mathsf{pk}_i = G^{\mathsf{sk}_i}$. The secret key can be reconstructed by combining a threshold $t = f + 1$ of the individual shares. We assume that pk_{smc} is registered on the blockchain of the access-control cothority.

Write transaction protocol

Wanda and each trustee of the access-control cothority perform the following protocol to log the write transaction tx_{w} on the blockchain. Wanda initiates the protocol through the following steps.

1. Retrieve the collective public key pk_{smc} of the secret-management cothority.

2. Choose a symmetric key k and encrypt the secret message m to be shared as $c_m = \text{enc}_k(m)$ and compute $H_{c_m} = \mathsf{H}(c_m)$. Set policy = pk_R to designate Ron as the intended reader of the secret message m.

3. Encrypt k towards pk_{smc} using a threshold variant of the ElGamal encryption scheme [229]. To do so, embed k as a point $k' \in \mathcal{G}$, pick a value r uniformly at random, compute $c_k = (\text{pk}_{\text{smc}}^r k', G^r)$ and create the NIZK proof π_{c_k} to guarantee that the ciphertext is correctly formed, CCA-secure and non-malleable.

4. Finally, prepare and sign the write transaction

$$\text{tx}_\text{w} = [c_k, \pi_{c_k}, H_{c_m}, \text{policy}]_{\text{sig}_{\text{sk}_W}}$$

   and send it to the access-control cothority.

The access-control cothority then logs the write transaction.

1. Verify the correctness of the ciphertext c_k using the NIZK proof π_{c_k}.

2. If the check succeeds, log tx_w in block b_w.

Read transaction protocol

After tx_w has been recorded, Ron needs to log a read transaction tx_r through the access-control cothority before he can request the decryption key shares. To do so, Ron performs the following steps.

1. Retrieve the ciphertext c_m and the block b_w, which stores tx_w, from the access-control cothority.

2. Check that $\mathsf{H}(c_m)$ is equal to H_{c_m} in tx_w to ensure that the ciphertext c_m of Wanda's secret has not been altered.

3. Compute $H_\text{w} = \mathsf{H}(\text{tx}_\text{w})$ as the unique identifier for the secret that Ron requests access to and determine the proof π_{tx_w} showing that tx_w has been logged on-chain.

4. Prepare and sign the read transaction

$$\text{tx}_\text{r} = [H_\text{w}, \pi_{\text{tx}_\text{w}}]_{\text{sig}_{\text{sk}_R}}$$

   and send it to the access-control cothority.

The access-control cothority then logs tx_r as follows.

1. Retrieve tx_w using H_w and use pk_R, as recorded in policy, to verify the signature on tx_r.
2. If the signature is valid and Ron is authorized to access the secret, log tx_r in block b_r.

Share retrieval protocol

After the read transaction has been logged, Ron can recover the secret data by running the share retrieval protocol with the secret-management cothority. To do so Ron initiates the protocol as follows.

1. Create and sign a secret-sharing request

$$\text{req}_\text{share} = [\text{tx}_\text{w}, \text{tx}_\text{r}, \pi_{\text{tx}_\text{r}}]_{\text{sig}_{\text{sk}_R}}$$

   where π_{tx_r} proves that tx_r has been logged on-chain.
2. Send req_share to each secret-management trustee to request the blinded shares.

Each trustee i of the secret-management cothority responds to Ron's request as follows.

1. Get G^r and pk_R from tx_w and prepare a blinded share $u_i = (G^r \, \text{pk}_R)^{\text{sk}_i}$ with a NIZK correctness proof π_{u_i}.
2. Create and sign the secret-sharing reply

$$\text{rep}_\text{share} = [u_i, \pi_{u_i}]_{\text{sig}_{\text{sk}_i}}$$

   and send it back to Ron.

Secret reconstruction protocol

To retrieve the decryption key k and recover the secret m, Ron performs as follows.

1. Wait to receive at least t valid shares $u_i = G^{(r+\text{sk}_R)\text{sk}_i} = G^{r'\text{sk}_i}$ and then use Lagrange interpolation to recover the blinded decryption key

$$\text{pk}_\text{smc}^{r'} = \prod_{k=0}^{t} (G^{r'\text{sk}_i})^{\lambda_i},$$

   where λ_i is the i^{th} Lagrange element.
2. Unblind $\text{pk}_\text{smc}^{r'}$ to get the decryption key pk_smc^{r} for c_k via

$$(\text{pk}_\text{smc}^{r'})(\text{pk}_\text{smc}^{\text{sk}_R})^{-1} = (\text{pk}_\text{smc}^{r})(\text{pk}_\text{smc}^{\text{sk}_R})(\text{pk}_\text{smc}^{\text{sk}_R})^{-1}$$

3. Retrieve the encoded symmetric key k' from c_k via

$$(c_k)(\mathrm{pk}_{\mathrm{smc}}^{r})^{-1} = (\mathrm{pk}_{\mathrm{smc}}^{r} k')(\mathrm{pk}_{\mathrm{smc}}^{r})^{-1},$$

decode it to k, and finally recover $m = \mathrm{dec}_k(c_m)$.

Ron may delegate the costly verification and combination of shares to a trustee, *i.e.*, the first step of the above protocol. The trustee is assumed to be honest-but-curious and to not DoS Ron. The trustee cannot access the secret, as he does not know sk_R and hence cannot unblind $\mathrm{pk}_{\mathrm{smc}}^{r'}$. Ron can detect if the trustee carries out the recovery incorrectly.

Evolution of the secret-management cothority

The secret-management cothority is expected to persist over a long period of time and to remain secure and available. However, a number of issues can arise over its lifetime. First, servers can join and leave the cothority resulting in churn. Second, even if the secret-management cothority membership remains static, the private shares of the servers should be regularly (*e.g.*, every month) refreshed to thwart an attacker who can attempt to collect a threshold of shares over a period of time. Lastly, the collective private key of the secret-management cothority should be rotated periodically *e.g.*, once every year or two. Any change of the current collective private-public key pair would require re-encrypting all of the long-lived secrets, however, if done by simply choosing a new key pair.

We address the first two problems by periodically re-sharing [129] the existing collective public key when a server joins or leaves the secret-management cothority, or when servers want to refresh their private key shares. Lastly, when the secret-management cothority wants to rotate the collective public/private key pair ($\mathrm{pk}_{\mathrm{smc}}$, $\mathrm{sk}_{\mathrm{smc}}$), CALYPSO needs to collectively re-encrypt each individual secret under the new collective public key. To achieve this, we can generate and use translation certificates [135] such that the secrets can be re-encrypted without the involvement of their writers and without exposing the underlying secrets to any of the servers.

Achieving system goals

Long-term secrets achieves its goals similarly to one-time secrets with these differences.

Confidentiality of Secrets. In long-term secrets, the secret message m is encrypted under a symmetric key k which is subsequently encrypted under a collective public key of the secret-management cothority such that at least $t = f + 1$ trustees must cooperate to decrypt it. The ciphertext is bound to a specific policy through the use of NIZK proofs [229] so it cannot be reposted in a new write transaction with a malicious reader listed in its policy. The access-control trustees log the write transaction tx_{w} that includes the encrypted key which, based on the properties of the encryption scheme, does not leak any information about k. After the secret-management trustees receive a valid request $\mathrm{req}_{\mathrm{share}}$, they respond with the blinded

shares of the collective private key encrypted under the public key in policy from the respective tx_w. Based on the properties of the DKG protocol, the collective private key is never known to any single entity and can only be used if t trustees cooperate. This means, only the intended reader gets a threshold of secret shares necessary to recover k and access m.

7.4.3 On-chain Blinded Key Exchange

In both on-chain secrets protocols, Wanda includes the public key of Ron in a secret's policy to mark him as the authorized reader. Once Wanda's write transaction is logged, everyone knows that she has shared a secret with Ron and correspondingly, once his read transaction is logged, everyone knows that he has obtained the secret. While this property is desirable for many deployment scenarios we envision, certain applications may benefit from concealing the reader's identity. See Section 7.7 for a discussion of CALYPSO's deployment.

We introduce an *on-chain blinded key exchange* protocol, an extension that can be applied to both on-chain secrets protocols. This protocol allows the writer to conceal the intended reader's identity in the write transaction and to generate a blinded public key for the reader to use in his read transaction. The corresponding private key can only be calculated by the reader and the signature under this private key is sufficient for the writer to prove that the intended reader created the read transaction. The protocol works as follows.

1. *Public Key Blinding.* Wanda generates a random blinding factor b and uses it to calculate a blinded version of Ron's public key $\mathsf{pk}_{\widetilde{R}} = \mathsf{pk}_R^b = G^{b\mathsf{sk}_R}$.

2. *Write Transaction.* Wanda follows either the one-time secrets or long-term secrets protocol to create tx_w with the following modifications. Wanda encrypts b under pk_R to enable Ron to calculate the blinded version of his public key by picking a random number b' and encrypting b as $(c_{b_1}, c_{b_2}) = (G^{\mathsf{sk}_R b'} b, G^{b'})$. Then, she uses $\mathsf{pk}_{\widetilde{R}}$ instead of pk_R in the policy. Wanda includes $c_b = (c_{b_1}, c_{b_2})$ and policy in tx_w. After tx_w is logged, she notifies Ron on a separate, secure channel that she posted tx_w such that he knows which transaction to retrieve.

3. *Read Transaction.* When Ron wants to read Wanda's secret, he first decrypts c_b using sk_R to retrieve $b = (c_{b_1})(c_{b_2}^{\mathsf{sk}_R})^{-1} = (G^{\mathsf{sk}_R b'} b)(G^{b'\mathsf{sk}_R})^{-1}$. Then, he can compute $\mathsf{sk}_{\widetilde{R}} = b\mathsf{sk}_R$ and use this blinded private key to anonymously sign his tx_r.

4. *Auditing.* If Wanda wants to prove that Ron generated the tx_r, she can release b. Then, anyone can unblind Ron's public key $\mathsf{pk}_R = \mathsf{pk}_{\widetilde{R}}^{-b}$, verify the signature on the transaction and convince themselves that only Ron could have validly signed the transaction as he is the only one who could calculate $\mathsf{sk}_{\widetilde{R}}$.

The on-chain blinded key exchange protocol enables Wanda to protect the identity of the intended reader of her secrets without forfeiting any of the on-chain secrets's guarantees,

however, it requires the knowledge of the reader's public key. As a consequence, this protocol does not support dynamic identities discussed in Section 7.5, as we cannot predict and blind an unknown, future public key. Nonetheless, this protocol provides a viable option for applications where relationship privacy is more important than dynamic identity evolution. An extension of this protocol that allows blinding of dynamic identities remains an open challenge to be addressed in future work.

7.4.4 Post-Quantum On-chain Secrets

The security of both on-chain secrets implementations relies on the hardness of the discrete logarithm (DL) problem. An efficient quantum algorithm [228] for solving the DL problem exists, however. One solution to provide post-quantum security in CALYPSO is to use post-quantum cryptography (*e.g.*, lattice-based cryptography [88]). Alternatively, we can implement on-chain secrets using the Shamir's secret sharing [227] scheme which is information-theoretically secure. Unlike the publicly-verifiable scheme we previously used, Shamir's secret sharing does not prevent a malicious writer from distributing bad secret shares because the shares cannot be verified publicly.

To mitigate this problem, we add a step to provide accountability of the secret sharing phase by (1) requiring the writer to commit to the secret shares she wishes to distribute and (2) requesting that each secret-management trustee verifies and acknowledges that the secret share they hold is consistent with the writer's commitment. As a result, assuming $n = 3f + 1$ and secret sharing threshold $t = f + 1$, the reader can hold the writer accountable for producing a bad transaction should he fail to correctly decrypt the secret message.

Write transaction protocol

Wanda prepares her write transaction tx_w with the help of the secret-management and access-control cothorities, where each individual trustee carries out the respective steps. Wanda initiates the protocol by preparing a write transaction as follows.

1. Choose a secret sharing polynomial $s(x) = \sum_{j=0}^{t-1} a_j x^j$ of degree $t-1$. The secret to be shared is $s = s(0)$.

2. Use $k = \mathsf{H}(s)$ as the symmetric key to compute the ciphertext $c = \text{enc}_k(m)$ for the secret message m and set $H_c = \mathsf{H}(c)$.

3. For each trustee i, generate a commitment $q_i = \mathsf{H}(v_i \parallel s(i))$, where v_i is a random salt value.

4. Specify the access policy and prepare and sign tx_w.

$$\text{tx}_\text{w} = [\langle q_i \rangle, H_c, \langle \mathsf{pk}_i \rangle, \text{policy}]_{\text{sig}_{\mathsf{sk}_W}}$$

5. Send the share $s(i)$, salt v_i, and tx_w to each secret-management trustee using a post-quantum secure channel.

The secret-management cothority verifies tx_w as follows.

1. Verify the secret share by checking that $(s(i), v_i)$ corresponds to the commitment q_i. If yes, sign tx_w and send it back to Wanda as a confirmation that the share is valid.

The access-control cothority finally logs Wanda's tx_w.

1. Wait to receive tx_w signed by Wanda and the secret-management trustees. Verify that at least $2f + 1$ trustees signed the transaction. If yes, log tx_w.

Read transaction, share request, and reconstruction protocols

The other protocols remain unchanged except that the secret-management trustees are already in possession of their secret shares and the shares need not be included in tx_r. Once Ron receives the shares from the trustees, he recovers the symmetric key k as before and decrypts c. If the decryption fails, then the information shared by Wanda (the key, the ciphertext, or both) was incorrect. Such an outcome would indicate that Wanda is malicious and did not correctly execute the tx_w protocol. In response, Ron can release the transcript of the tx_r protocol in order to hold Wanda accountable.

7.5 Skipchain Identity and Access Management

The CALYPSO protocols described so far do not provide dynamic sovereign identities. They only support static identities (public keys) and access policies as they provide no mechanisms to update these values. These assumptions are rather unrealistic though, as the participants may need to change or add new public keys to revoke a compromised private key or to extend access rights to a new device, for example. Similarly, it should be possible to change access policies so that access to resources can be extended, updated or revoked and to define access-control rules for individual identities and groups of users for greater flexibility and efficiency. Finally, any access-control system that supports the above properties should be efficient as well as secure to prevent freeze and replay attacks [195], and race conditions between applying changes to access rights and accessing the resources.

In order to address these challenges and achieve dynamic sovereign identities, we introduce the *skipchain-based identity and access management* (SIAM) subsystem for CALYPSO that provides the following properties: (1) Enable users to specify and announce updates to resource access keys and policies. (2) Support identities for both individual users and groups of users.

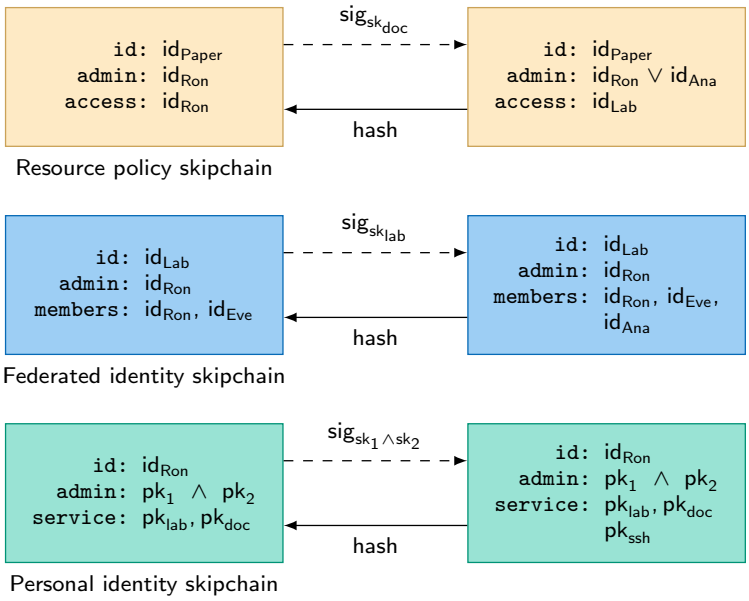

Figure 7.3 – Skipchain-based identity and access management (SIAM): First, Ron updates his personal identity skipchain id_{Ron} to include pk_{ssh}. Afterwards, he uses sk_{lab} to extend the federated identity skipchain id_{lab} to add id_{Ana} as a member. Finally, he adds id_{Ana} as an administrator and id_{lab} as authorized readers to the resource policy skipchain id_{paper} by using sk_{doc}.

(3) Protect against replay and freeze attacks. (4) Enforce atomicity of accessing resources and updating resource access rights to prevent race conditions.

7.5.1 Architecture

In SIAM, we introduce three types of skipchains [195], structures similar to a blockchain but doubly-linked. *Personal identity skipchains* store the public keys that individual users control and use to access resources. A user can maintain a number of public keys that correspond to his identity that are used for access to resources on different devices, for example. *Federated identity skipchains* specify identities and public keys of a collective identity that encompasses users that belong to some group, such as employees of a company, members of a research lab, a board of directors, etc. *Resource policy skipchains* track access rights of identities, personal or federated, to certain resources and enable dynamic access control. In addition to listing identities and their public keys, policy skipchains include access-control rules allowing to enforce fine-grained update conditions for write transactions. Section 7.8 describes a simple access-control list (ACL) we created for our implementation.

The main insight is that skipchains securely maintain a verifiable timeline of changes to the identities and policies they represent. This means that these identities or policies can evolve dynamically and independently of the specific applications that make use of them and that at

any point the applications can verifiably obtain the most up-to-date version of each.

The SIAM skipchains are under the self-sovereign control of individual users or groups of users. To track administrative rights, each SIAM skipchain includes in its skipblocks the (public) administrative keys of users authorized to make updates and the policy under which a skipchain can be updated. These update policies can be expressed as arbitrary boolean circuits, *e.g.,* requiring approvals from just a single administrator or a certain set of them. Since the administrative keys are used only for skipchain updates, they should be stored in cold wallets, such as hardware security modules, for increased security.

To evolve a SIAM skipchain and consequently the identities or access policies it represents, its administrators follow the skipchain's update policies and create a new skipblock that reflects the necessary changes, and then publicly announce it, *e.g.,* by pushing the update to the storage provider(s) maintaining a public interface to the SIAM skipchain. Users and services that follow SIAM skipchains can get notified automatically about those updates and accept them if they are proposed by authorized users and adhere to the update policies. Since the latest skipblock represents the current state of a skipchain, *i.e.,* identities of all currently authorized users or all current access rules, revocation is trivially supported as the administrators simply push a new skipblock to the respective skipchain that omits the public key or access rule that needs to be revoked. Figure 7.3 provides an overview on SIAM.

7.5.2 Integration Into CALYPSO

To integrate SIAM with CALYPSO, the long-term secrets protocols described in Section 7.4.2 are adapted as follows. Assume that Ron has logged the unique identifier id_R of his personal identity skipchain on the access-control blockchain. If Wanda wants to give Ron access to a resource, she simply sets policy = id_R instead of policy = pk_R in $\mathrm{tx_w}$.

This means that instead of defining access rights in terms of Ron's static public pk_R, she does so in terms of Ron's skipchain and consequently, any public key(s) specified in the most current block of id_R. Then, the resource is encrypted under the shared public key of the secret-management cothority as before. To request access, Ron creates the read transaction

$$\mathrm{tx_r} = [H_w, \pi_{\mathrm{tx_w}}, \mathsf{pk}_{R'}]_{\mathrm{sig}_{\mathsf{sk}_{R'}}}$$

where $H_w = \mathsf{H}(\mathrm{tx_w})$ is the unique identifier for the secret that Ron requests access to, $\pi_{\mathrm{tx_w}}$ is the blockchain inclusion proof for $\mathrm{tx_w}$, and $\mathsf{pk}_{R'}$ is one of Ron's public keys that he wishes to use from the latest block of the id_R skipchain. After receiving $\mathrm{tx_r}$, the access-control cothority follows the id_R skipchain to retrieve the latest skipblock and verifies $\mathsf{pk}_{R'}$ against it. Then, the access-control cothority checks the signature on $\mathrm{tx_r}$ using $\mathsf{pk}_{R'}$ and, if valid, logs $\mathrm{tx_r}$. Once $\mathrm{tx_r}$ is logged, the rest of the protocol works as described in Section 7.4.2, where the secret-management cothority uses $\mathsf{pk}_{R'}$ for re-encryption to enable Ron to retrieve the resource.

7.5.3 Achieving SIAM Goals

When SIAM is used, Ron is able to evolve the id_R skipchain arbitrarily, *e.g.*, rotate existing access keys or add new devices, and still retain access to the encrypted resource without needing Wanda to update the initial write transaction. Analogously, Wanda can efficiently provide a group of users access to a resource by using a federated identity id_F that these users are a part of, instead of adding each user individually, by setting policy = id_F in tx_w. This approach outsources the resource access maintenance to the administrators of the id_F skipchain as they are in charge of the federated identity's membership and can add and remove members at will. Alternatively, Wanda can set up a resource policy skipchain id_P she is in charge of and include id_F as non-administrative members along with any other rules she wants to have enforced. Then, Wanda would use policy = id_P in tx_w authorizing id_F to access the respective resource under the specified rules.

In order for users to defend against freeze and replay attacks we require them to generate freshness proofs of their SIAM skipchains. To do this they submit in regular time periods (*e.g.*, every hour) the head of their skipchain for timestamping on the blockchain. This prevents freeze and replay attacks as an adversary that managed to subvert an old SIAM skipblock cannot convince a client to accept the adversarial actions as authoritative, given that the skipblock the adversary refers to is different from the fresh one appearing on the blockchain. This practice further ensures the atomicity of read, write, and (skipchain) update operations, at the moment a SIAM update happens the client should send the new SIAM skipblock for timestamping. This effectively serializes reads, writes, and updates and therefore prevents race conditions.

7.6 Further Security Consideration

Our contributions are mainly pragmatic rather than theoretical as we employ only existing, well-studied cryptographic algorithms. We already discussed achieving CALYPSO's security goals in the previous sections. On-chain secrets protocols achieve all goals but dynamic sovereign identities which is addressed by SIAM. In this section, we discuss the effect of malicious parties on CALYPSO.

Malicious readers and writers CALYPSO's functionality resembles a fair-exchange protocol [197] in which a malicious reader may try to access a secret without paying for it and a malicious writer may try to get paid without revealing the secret. In CALYPSO, we protect against such attacks by employing the access-control and secret-management cothorities as decentralized equivalents of trusted third parties that mediate interactions between readers and writers.

The access-control cothority logs a write transaction on the blockchain only after it successfully verifies the encrypted data against the corresponding consistency proof. This ensures that

a malicious writer cannot post a transaction for a secret that cannot be recovered. Further, as each tx_w binds its contents to its policy, it protects against attacks where malicious writers naively extract contents of already posted transactions and submit them with a different policy listing themselves as the authorized readers. Similarly, before logging a read transaction, the access-control cothority verifies that it refers to a valid tx_w and it is sent by an authorized reader as defined in the policy of tx_w. A logged read transaction serves as an access approval. The secret-management cothority releases the decryption shares to the authorized reader only after confirming the reader presents an auditable proof of tx_r.

Malicious trustees Our threat model permits a fraction of the access-control and secret-management cothority trustees to be dishonest. The thresholds ($t = f + 1$) used in on-chain secrets, however, prevent the malicious trustees from being able to pool their secret shares and access writers' secrets or to prevent an authorized reader from accessing their secret by withholding the secret shares. Further, even if some individual malicious trustees refuse to accept requests from the clients or to participate in the protocols altogether, the remaining honest trustees are able to carry out all protocols by themselves thereby ensuring service availability.

Malicious storage providers Wanda may choose to store the actual encrypted data either on-chain or off-chain by choosing to outsource the storage to external providers. Because the data is encrypted, it can be shared with any number of possibly untrusted providers. Before Ron creates a tx_r he needs to retrieve and verify the encrypted data against the hash posted in tx_w. If Ron cannot obtain the encrypted data from the provider, he can contact Wanda to expose the provider as dishonest and receive the encrypted data directly from Wanda or an alternative storage provider.

7.7 Experience Using CALYPSO

Below we describe three real-world deployments, two completed and one in-progress, of CALYPSO that resulted from collaborations with companies that needed a flexible, secure, and decentralized solution to share data. We also describe a zero-collateral, constant-round decentralized lottery and compare it with the existing solutions.

7.7.1 Auditable Online Invoice Issuing

Together with Conextrade[2], the main invoice regulator of Switzerland, we built an auditable online invoice issuing system. It uses HyperLedger Fabric v1.0 as the access-control blockchain together with long-term secrets. While the system uses static identities, they are blinded as needed using the protocol described in Section 7.4.3.

[2] https://www.conextrade.com/

Problem definition The system consists of a set of potentially mutually distrustful sellers and buyers as well as a regulator, who are all part of a dynamic ecosystem. To keep track of all business relationships without the need for an intermediary the system relies on blockchain technology. A seller wishes to verifiably issue an invoice to a buyer while granting additional access to the regulator. The invoice contains confidential information that both parties want to protect. The goal was to allow the invoices to be logged and tracked and to enable the regulator to access the details if an issue arises between the parties.

Solution with CALYPSO This system required a straightforward deployment of CALYPSO. The sellers generate write transactions where the secret message is the invoice and the authorized readers are both the buyer and the regulator. When the buyer sees the respective write transaction, he issues a read transaction to access the invoice. If there is an issue with the invoice or no invoice has been issued for a certain amount of time, the buyer reports it to the regulator who can audit the transactions and the invoice. Analogously, the seller can request the regulator to audit if an issue arises on his side. Using CALYPSO's blinded identities in the write transactions hides the relationships between the sellers and buyers and consequently details such as trade frequencies and types of purchases, which is advantageous from a business perspective.

7.7.2 Clearance-enforcing Document Sharing

We have used CALYPSO to deploy a decentralized, clearance-enforcing document-sharing system that enables two organizations, A and B, to share a document D, such that a policy of confidentiality can be enforced on D. We have realized this system with ByzGen[3] a contract of the Ministry of Defense of the UK using ByzCoin (Chapter 3) and long-term secrets. The evaluation of this application is in Section 7.9.2.

Problem Definition Organization A wants to share with organization B a document D whose entirety or certain parts are classified as confidential and should only be accessible by people with proper clearance. Clearance is granted to (or revoked from) employees individually as needed or automatically when they join (or leave) a specific department so the set of authorized employees continuously changes. The goal is to enable the mutually distrustful A and B to share D while dynamically enforcing the specific clearance requirements and securely tracking accesses to D for auditing.

Solution with CALYPSO First, A and B agree on a mutually-trusted blockchain system to implement the access-control cothority whose trustees include servers controlled by both organizations. Then, each organization establishes federated identity skipchains with all the identities that have clearance, id_A and id_B, respectively which include references to: (a)

[3] https://byzgen.com/

federated skipchains for departments that have a top-secret classification (*e.g.*, senior management), (b) federated skipchains for attributes that have a top-secret classification (*e.g.*, ranked as captain) and (c) personal skipchains of employees that need exceptional clearance.

Organization A creates a document D, labels each paragraph as confidential or unclassified and, encrypts it using a different symmetric key. A shares the ciphertext with B and generates tx_w, which contains the symmetric keys of the classified paragraphs and policy = id_B. Any employee of B whose public key is included in the set of classified employees as defined in the most current skipblock of id_B can retrieve the symmetric keys by creating read transactions. CALYPSO logs the tx_r creates a proof of access and delivers the key. Both organizations can update their identity skipchains as needed to ensure that at any given moment only authorized employees can access.

7.7.3 Patient-centric Medical Data Sharing

CALYPSO lends itself well for applications that require secure data-sharing for research purposes [211]. We are in the process of working with hospitals and research institutions from a Switzerland as part of the DPPH project[4] to build a patient-centric system to share medical data. We expect to use OmniLedger (Chapter 4) along with long-term secrets. We do not provide an evaluation of this application as it is similar to the previous one.

Problem Definition Researchers face difficulties in gathering medical data from hospitals as patients increasingly refuse to approve access to their data for research purposes amidst rapidly-growing privacy concerns [130]. Patients dislike consenting once and completely losing control over their data and are more likely to consent to sharing their data with specific institutions [144]. The goal of this collaboration is to enable patients to remain sovereign over their data; hospitals to verifiably obtain patients' consent for specific purposes; and researchers to obtain access to valuable patient data. In the case that a patient is unable to grant access (unconscious), the medical doctor can request an exception (specified in the policy) and access the data while leaving an auditable proof.

Solution with CALYPSO We have designed a preliminary architecture for a data-sharing application that enables a patient P to share her data with multiple potential readers over time. This deployment is different from the previously described one in that the data generator (hospital) and the data owner (P) are different. For this reason, we use a resource policy skipchain id_P such that the hospital can represent P's wishes with respect to her data. Policy skipchains can dynamically evolve by adding and removing authorized readers, and can include rich access-control rules.

CALYPSO enables P to initialize id_P when she first registers with the medical system, Initially,

[4] https://dpph.ch/

id_P is empty, indicating that P's data cannot be shared. If a new research organization or other hospital requests to access some of P's data, then P can update id_P by adding a federated identity of the research organization and specific rules. When new data is available for sharing, the hospital generates a new write transaction that consists of the encrypted and possibly obfuscated, or anonymized medical data and id_P as policy. As before, users whose identities are included in id_P can post read transactions to obtain access. Hence, P remains in control of her data and can unilaterally update or revoke access.

7.7.4 Decentralized Lottery

Problem Definition We assume there is a set of n participants who want to run a decentralized zero-collateral lottery selecting one winner. The lottery is managed by a smart contract that collects the bids and waits for the final randomness to decide on the winner. The evaluation of this application is in Section 7.9.3.

Solution with CALYPSO Each participant creates a tx_w where the secret is their contribution to the randomness calculation and shares it using long-term secrets. After a predefined number of blocks (the barrier point) the input phase of the lottery closes. Afterwards the smart contract creates a tx_r to retrieve all inputs submitted before the barrier point and posts the reconstructed values and the corresponding proofs.

Once the final randomness has been computed as an XOR of all inputs, the smart contract uses it to select the winner.

Comparison to Existing Solutions Prior proposals for decentralized lotteries either need collateral (*e.g.*, Ethereum's RandDAO [212]) or run in a non-constant number of rounds [183]. CALYPSO enables a simpler decentralized lottery design, as the lottery executes in one round and needs no collateral because the participants cannot predict the final randomness or abort prematurely.

7.8 Implementation

We implemented all components of CALYPSO, on-chain secrets, and SIAM, in Go [118]. For cryptographic operations, we relied on Kyber [160], an advanced crypto library for Go. In particular, we used its implementation of the Edwards25519 elliptic curve, which provides a 128-bit security level. For the consensus mechanism required for the access-control cothority, we used a publicly available implementation of ByzCoin [149], a scalable Byzantine consensus protocol. We implemented both on-chain secrets protocols, one-time and long-term secrets, run by the secret-management cothority. For SIAM, we implemented signing and verifying using a JSON-based ACL as described below. All of our implementations are available under

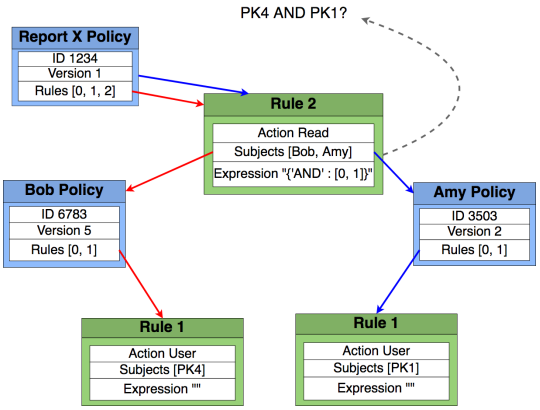

Figure 7.4 – Verifier's path checking for multi-signature requests.

```
{
    "ID" :  2345
    "Version" : 1,
    "Rules" :
    [
        {
            "Action" : "Admin",
            "Subjects" : [S1, S2],
            "Expression" : "{'AND' : [S1, S2]}"
        }
    ]
}
```

Figure 7.5 – Sample Policy in JSON access-control language.

an open source license on GitHub.

We used a simple JSON-based access-control language to describe policies in CALYPSO, however, different deployments of CALYPSO might benefit from more expressive ACLs. A policy consists of a unique identifier, a version number, and a list of rules that regulate the access to secrets stored in tx_w. A rule has the following three fields. An `action` field which refers to the activity that can be performed on the secret (*e.g.*, `READ` or `UPDATE`). A `subjects` field listing the identities (*e.g.*, id_Ron) that are permitted to perform the `action`. Lastly, an `expression` field which is a string of the form `operator` : [`operands`], where the `operator` is a logical operation (`AND` and `OR` in our case) and `operands` are either `subjects` or other `expressions` that describe the conditions under which the rule can be satisfied. More concretely, a sample expression could be $\{\texttt{AND} : [\text{id}_\text{Lab}, \text{id}_\text{Ron}]\}$, which means that signatures of both id_Lab and id_Ron are required to satisfy that rule. To express more complex conditions we can nest expressions, for example $\{\texttt{OR} : [\{\texttt{AND} : [\text{id}_1, \text{id}_2]\}, \{\texttt{AND} : [\text{id}_3, \text{id}_4]\}]\}$ evaluates to $((\text{id}_1\ \texttt{AND}\ \text{id}_2)\ \texttt{OR}\ (\text{id}_3\ \texttt{AND}\ \text{id}_4))$. We describe single and multi-signature access requests against policies and outline how they are created and verified below.

7.8.1 Access Requests and Verification

In this section, we outline how we create and verify access requests. A request consists of the policy and the rule invoked that permits the requester to perform the action requested. There is also a message field where extra information can be provided *e.g.*, a set of documents is governed by the same policy but the requester accesses one specific document. A request req is of the form: $\text{req} = [\text{id}_{Policy}, \text{index}_{Rule}, M]$, where id_{Policy} is the ID of the target policy outlining the access rules; index_{Rule} is the index of the rule invoked by the requester; and M is a message describing extra information.

To have accountability and verify that the requester is permitted to access, we use signatures. The requester signs the request and creates a signature consisting of the signed request (sig_{req}) and the public key used (pk). On receiving an access request, the verifier checks that the sig_{req} is correct. The verifier then checks that there is a valid path from the target policy, id_{Policy}, to the requester's public key, pk. This could involve multiple levels of checks, if the requester's key is not present directly in the list of *subjects* but included transitively in some federated SIAM that is a *subject*. The verifier searches along all paths (looking at the last version timestamped by the access-control cothority) until the requester's key is found.

Sometimes, an access request requires multiple parties to sign. Conditions for multi-signature approval can be described using the *expression* field in the rules. An access request in this case would be of the form (req, [sig_{req}]) where [sig_{req}] is a list of signatures from the required-for-access parties. The verification process is similar to a single signature case.

Figure 7.4 shows an example of the path verification performed by the verifier. Report X has a policy with a Rule granting read access to Bob and Amy. There is an expression stating that both Bob's and Amy's signatures are required to obtain access. Hence, if Bob wants access, he sends a request (req, [sig_{req,pk_1}, sig_{req,pk_4}]), where req = [1234, 2, "ReportX"] The verifier checks the paths from the policy to Bob's pk_4 and Amy's to pk_1 are valid. Paths are shown in red and blue respectively. Then the expression 'AND' : [0,1] is checked against the signatures. If all checks pass, the request is considered to be verified.

7.8.2 JSON Access-Control Language

A sample policy for a document, expressed in the JSON based language, is shown in Figure 7.5. The policy states that it has one Admin rule. The admins are S1 and S2 and they are allowed to make changes to the policy. The Expression field indicates that any changes to the policy require both S1 and S2's signatures.

7.9 Evaluation

To evaluate CALYPSO, we use micro-benchmarks to compare on-chain secrets against both a centralized and a semi-centralized solution using simulated workloads. We also evaluate CALYPSO using simulated and real data traces in the context of two applications: clearance-enforcing document sharing (see Section 7.7.2) and a decentralized lottery (see Section 7.7.4). We remark that the synthetic workloads we generated were significantly heavier than those from the real data traces. We ran all our experiments on 4 Mininet [187] servers, each equipped with 256 GB of memory and 24 cores running at 2.5 GHz. To simulate a realistic network, we configured Mininet with a 100 ms point-to-point latency between the nodes and a bandwidth of 100 Mbps for each node.

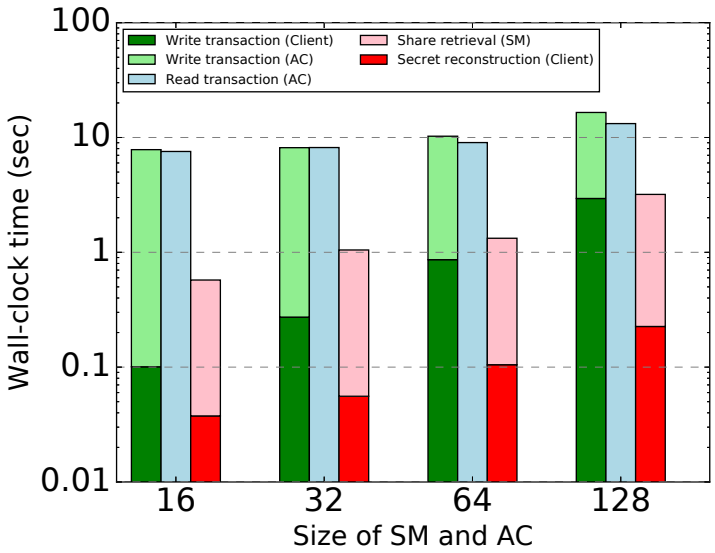

Figure 7.6 – Latency of one-time secrets protocol for varying sizes of the secret-management (SM) and access-control (AC) cothorities.

7.9.1 Mirco-benchmarks

We evaluated and compared both on-chain secrets protocols as well as the overheads introduced through the use of the dynamic identities and policies of SIAM. The primary questions we wish to investigate for on-chain secrets are whether its latency overheads are acceptable when deployed on top of blockchain systems and whether it can scale to hundreds of validators as required to ensure a high degree of confidence in the security of the system. In all experiments, we checked the time it takes to create read and write transactions given different sizes of secret-management and access-control cothorities. For SIAM, we evaluate the latency overhead of creating and verifying access requests against SIAM skipchains both for simple identities as well as for complex policies.

On-chain Secrets

In our experiments, we measure the overall latency of both on-chain secrets protocols, as shown in Figure 7.2, where we investigate the cost of the write, read, share retrieval and share reconstruction sub-protocols. In the experiments, we vary the number of trustees to determine the effects on the latency and we remark that in our implementation all trustees are part of both cothorities.

One-time secrets Figure 7.6 shows the latency results for varying sizes of access-control and secret-management cothorities. First, we observe that the client-side creation of the tx_w is a costly operation which takes almost one second for 64 secret-management trustees. This is expected as preparing the tx_w involves picking a polynomial and evaluating it at n points, and

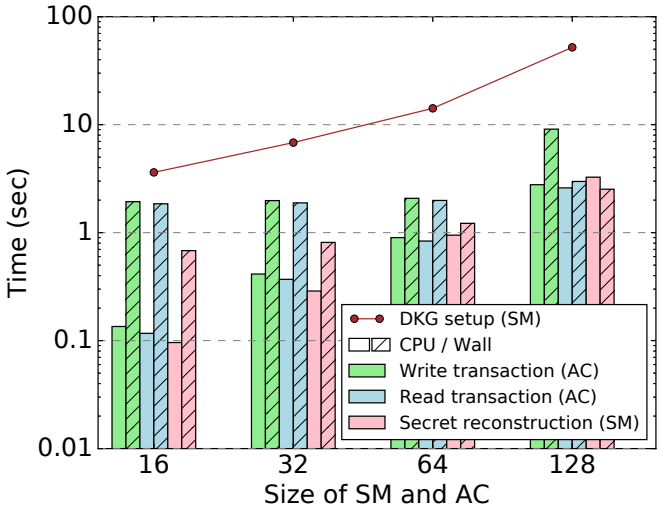

Figure 7.7 – Latency of long-term secrets protocol for varying sizes of secret-management (SM) and access-control (AC) cothorities.

setting up the PVSS shares and commitments, all of which involve expensive ECC operations. Second, we observe that the tx_w and tx_r processing times at the access-control cothority are comparable, but a write takes ≈ 250 ms longer on average than a read. This is due to the fact that the access-control trustees verify all NIZK encryption proofs. Our experiments also show that verifying the NIZK decryption proofs and recovering the shared secret are substantially faster than creating the tx_w and differ by an order of magnitude for large numbers of shares (*e.g.*, for 128 shares, ≈ 250 ms vs ≈ 3 sec). This is due to the fact that verifying the NIZK proofs and reconstructing the shared secret require less ECC computations than the computationally expensive setup of the PVSS shares. Finally, we observe that the overhead for the secret-management cothority part of the secret recovery is an order of magnitude higher than the client side. This is expected as the client sends a request to each secret-management trustee and waits until a threshold of them replies.

Long-term secrets Figure 7.7 presents the overall latency costs of the cothority setup (DKG), write, read, share retrieval and share reconstruction sub-protocols. Except for the DKG setup, all steps of the long-term secrets protocol scale linearly in the size of the cothority. Even for a large cothority of 128 servers, it takes less than 8 seconds to process a write transaction. The CPU time is significantly lower than the wall-clock time due to the network (WAN) overhead that is included in the wall-clock measurements. While the DKG setup is quite costly, especially for a large number of servers, it is a one-time cost incurred only at the start of a new epoch. The overhead of the share retrieval is linear in the secret-management cothority as the number of shares t, which need to be validated and interpolated, increases linearly in the size of the secret-management cothority.

The size of transactions is smaller in long-term secrets than in one-time secrets because the

Table 7.1 – tx_w size for varying secret-management cothority sizes

	tx_w size (bytes)	
Number of trustees	One-time secrets	Long-term secrets
16	4'086	160
32	8'054	160
64	15'990	160
128	31'926	160

data is encrypted under the secret-management's collective public key which results in a constant overhead regardless of the cothority's size. Table 7.1 shows tx_w sizes in one-time secrets and long-term secrets for different secret-management cothority configurations. In one-time secrets, a tx_w stores three pieces of PVSS-related information: encrypted shares, polynomial commitments and NIZK encryption consistency proofs. As the size of this information is determined by the number of PVSS trustees, the size of the tx_w increases linearly with the size of the secret-management cothority. In long-term secrets tx_w uses the collective key of the secret-management cothority and does not need to include the encrypted shares. As a result, long-term secrets have constant write transaction size.

Skipchain-based Identity and Access Management

For SIAM, we benchmark the cost of validating the signature on a read transaction which is the most resource and time intensive operation. We distinguish single and multi-signature requests. The single signature case represents simple requests where one identity is requesting access while multi-signature requests occur for more complex access-control rules.

Single-signature request verification For single-signature requests, the verification time is the sum of the signature verification and the time to validate the identity of the reader requesting access by checking it against the identity of the target reader as defined in the policy. The validation is done by finding the path from the target's skipchain to the requester's skipchain. We vary the *depth* of the requester, which refers to the distance between the two skipchains. Figure 7.8 shows the variation in request verification time depending on the requester's depth. We observe that most of the request verification time is required for signature verification which takes $\approx 385\mu s$ and accounts for 92.04–99.94% of the total time.

Multi-signature request verification As signature verification is the main overhead, we investigate the effect of verifying multi-signature requests. We create requests with a varying number of signers and investigate the number of request per second we can verify. Figure 7.9 shows the results for a requester skipchain's depth of 2 and 10. There is a significant reduction in the number of requests that can be verified when the number of signers increases whereas

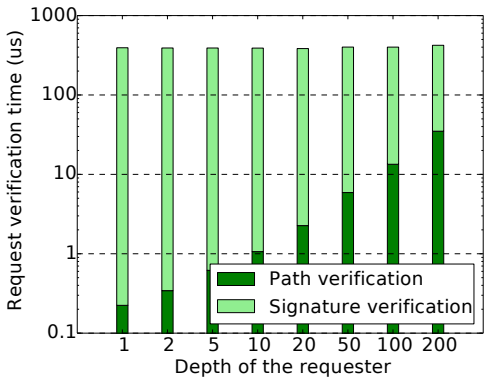

Figure 7.8 – Single-signature request verification.

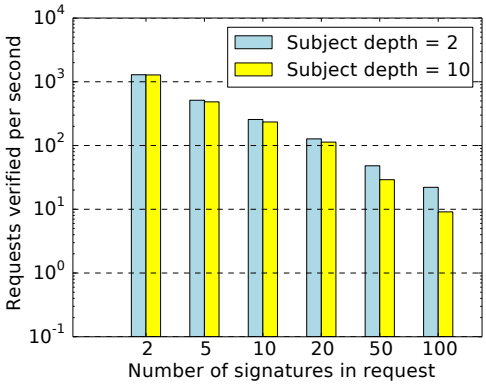

Figure 7.9 – Multi-signature request verification throughput.

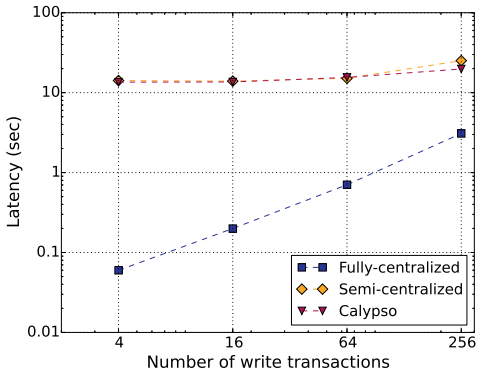

Figure 7.10 – Write transaction latency for different loads in clearance-enforcing document sharing.

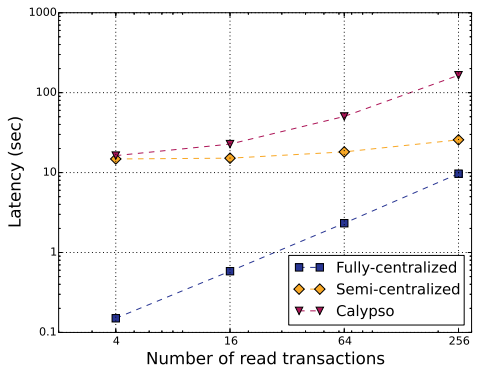

Figure 7.11 – Read transaction latency for different loads in clearance-enforcing document sharing.

the depth of the requester is not significant.

7.9.2 Clearance-Enforcing Document Sharing

We compare the clearance-enforcing document sharing deployment of CALYPSO with both a fully-centralized access-control system (one cloud server manages everything) and a semi-centralized one where accesses and policies are logged on-chain but the data is managed in the cloud. We vary the simulated workload per block from 4 to 256 concurrent read and write transactions. Figure 7.10 shows that CALYPSO takes 5× to 100× more time to execute a write transaction compared to the centralized solution and has the same overhead as the semi-centralized system. Figure 7.11 shows that CALYPSO takes 10× to 100× more time to execute a read transaction compared to the centralized solution, but it only takes 0.2× to 5× more time than the semi-centralized system. These experiments use a blockchain with a blocktime of 7 seconds. For a slower blockchain, such as Bitcoin, the overall latency of all

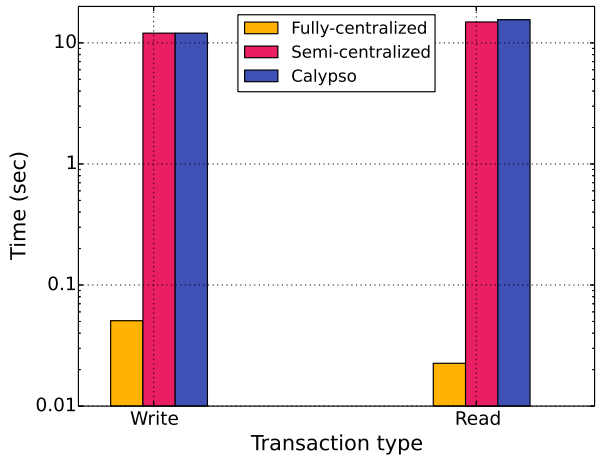

Figure 7.12 – Average write and read transaction latencies replaying the real-world data traces from the clearance-enforcing document sharing deployment

transactions would be dominated by the (longer) blocktime.

Next, we evaluate the clearance-enforcing document sharing deployment of CALYPSO using real-world data traces from our governmental contractor partner mentioned in Section 7.7.2. Data traces are collected from the company's testbed over a period of 15 days. There are 1821 tx_w and 1470 tx_r, and the minimum, maximum and average number of transactions per block are 1, 7 and 2.62, respectively. We replayed the traces on CALYPSO, as well as our fully-centralized and semi-centralized access-control system implementations. We use a blocktime of 10 seconds as it is in the original data traces. Figure 7.12 shows the average latency for the write and read transactions. The results show that CALYPSO and the semi-centralized system have comparable read transaction latencies as it is dominated by the blocktime, which agrees with our micro-benchmark results.

7.9.3 Decentralized Lottery

We compared our CALYPSO-based zero-collateral lottery with the corresponding lottery by Miller et al. [183] (tournament) simulated and real workloads. Figure 7.14 shows that CALYPSO-based lottery performs better both in terms of the overall execution time and necessary bandwidth. Specifically, our lottery runs in one round (it always takes two blocks to finish the lottery) while the tournament runs in a logarithmic number of rounds due to its design consisting of multiple two-party lotteries.

Next, we evaluate both lottery implementations using the transactions from Fire Lotto [72], an Ethereum-based lottery, see Figure 7.13 for overall time comparisons. We considered transactions sent to the Fire Lotto smart contract over a period of 30 days and each data point in the graph corresponds to a single lottery run. As before, CALYPSO-based lottery

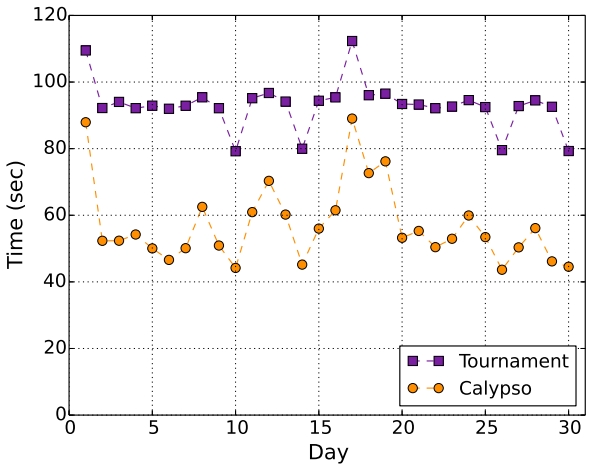

Figure 7.13 – CALYPSO vs Tournament lottery using Fire Lotto workloads.

performs better because it completes in one round whereas the tournament lottery requires a logarithmic number of interactions with the blockchain and consequently has a larger overhead. More specifically, while the blocktime of 15 seconds makes up 14–20% of the total latency in CALYPSO, it contributes most of the per-round latency to the tournament lottery. We remark that our results only include the latency of the reveal phase since the commit phase happens asynchronously over a full day.

7.10 Conclusion

We have presented CALYPSO, the first fully decentralized framework for auditable access control on protected resources over a distributed ledger that maintains confidentiality and control of the resources even after they have been shared. CALYPSO achieves its goals by introducing two separate components. The first component, on-chain secrets, is deployed on top of a blockchain to enable transparent and efficient management of secret data via threshold cryptography. The second component, skipchain-based identity and access management, allows for dynamic identities and resource access policies. We have implemented CALYPSO and shown that it can be efficiently deployed with blockchain systems to enhance their functionality. Lastly, we described four deployments of CALYPSO to illustrate its applicability to real-world applications.

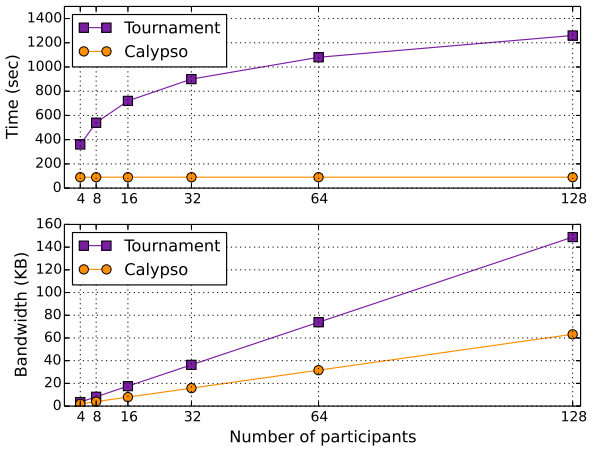

Figure 7.14 – CALYPSO vs Tournament lottery using simulated workloads.

8 Horizontal Scaling and Confidentiality on Permissioned Blockchains

8.1 Introduction

In this chapter, we look into enabling sharding in the permissioned setting, where the adversarial power can be relaxed. First, we deploy channels for horizontal scaling drawing inspiration from the state of the art [150, 74], but at the same time navigating the functionality and trust spectrum to create simplified protocols with less complexity and need for coordination. Then, we introduce the idea that, in a permissioned setting, we can leverage the state partition that a channel introduces as a confidentiality boundary. In the second part of the chapter, we show how we enable confidential channels while preserving support for cross-shard transactions.

Our main contributions are (a) the support for horizontal scaling on permissioned blockchains with cross-channel transaction semantics, (b) the use of channels as a confidentiality boundary and (c) the formalization of an asset-management application on top of blockchain systems.

8.2 Preliminaries

Blockchain Definitions In the context of this work, a blockchain is an append-only tamper-evident log maintained by a distributed group of collectively trusted nodes. When these nodes are part of a defined set [10], we call the blockchain *permissioned*. Inside every block, there are transactions that may modify the state of the blockchain (they might be invalid [10]). A distributed ledger [240] is a generalization of a blockchain as it can include multiple blockchains that interact with each other, given that sufficient trust between blockchains exists.

We define the following roles for nodes in a blockchain:

- **Peers** execute and validate transactions. Peers store the blockchain and need to agree on the state.
- **Orderers** collectively form the ordering service. The ordering service establishes the total order of transactions. Orderers are unaware of the application state and do not

participate in the execution or validation of transactions. Orderers reach consensus [10, 149, 191, 50] on the blocks in order to provide a deterministic input for the blockchain peers to validate transactions.

- **Oracles** are special nodes that provide information about a specific blockchain to nodes not being peers of that blockchain. Oracles come with a *validation policy* of the blockchain defining when the announcement of an oracle is trustworthy[1].
- **(Light) Clients** submit transactions that either read or write the state of a distributed ledger. Clients do not directly subscribe to state updates, but trust some oracles to provide the necessary proofs that a request is valid.

Nodes can implement multiple roles or collapse roles (*e.g.,* miners in Bitcoin [191] are concurrently peers and orderers). In a distributed ledger that supports multiple blockchains that interoperate the peers of one blockchain necessarily implement a client for every other blockchain and trust the oracles to provide proofs of validity for a cross-channel transaction. A specific oracle instantiation can be for example that a quorum (*e.g.,* $\frac{2}{3}$) of the peers need to sign any announcement for it to be valid.

8.2.1 Channels

In this paper we extend channels (first introduced in Hyperledger Fabric [10]), an abstraction similar to shards. In prior work [10], a channel is defined as an autonomous blockchain agnostic to the rest of the state of the system. In this work, we redefine a channel as a state partition of the full system that (a) is autonomously managed by a (logically) separate set of peers (but is still aware of the bigger system it belongs) and (b) optionally hides the internal state from the rest of the system.

A channel might communicate with multiple other channels; and there needs to be some level of trust for two channels to transact. Hence, we permit each channel to decide on what comprises an authoritative proof of its own state. This is what we call **validation policy**: clients need to verify this policy in order to believe that something happened in a channel they are transacting with. When channel A wants to transact with channel B, then the peers of A effectively implement a client of channel B (as they do not know the state of B directly). Thus, the peers of A verify that the validation policy of B is satisfied when receiving authoritative statements from channel B.

For channels to interact, they need to be aware of each other and be able to communicate. Oracles are responsible for this functionality, as they can gossip authoritative statements (statements supported by the validation policy) to the oracles of the other channels. This functionality needs a bootstrap step where channels and validation policies are discovered, which we do not address in this work. A global consortium of organizations could publicly

[1] *e.g.,* in Bitcoin the oracles will give proofs that have 6 Proofs-of-Work build on top of them

announce such information, or consortia represented by channels could communicate off-band. Once a channel is established further evolution can be done without a centralized intermediary, by using skipchains [195].

8.2.2 Threat Model

The peers that have the right to access one channel's state are trusted for confidentiality, meaning that they will not leak the state of the channel on purpose. We relax this assumption later, providing forward and backward secrecy in case of compromise. We assume that the ordering service is secure, produces a unique blockchain without forks and the blocks produced are available to the peers of the channels. We further assume that the adversary is computationally bounded and that cryptographic primitives (*e.g.,* hash functions and digital signatures) are secure.

8.2.3 System Goals

We have the following primary goals.

1. **Secure transactions.** Transactions are committed atomically or eventually aborted, both within and across channels.

2. **Scale-out.** The system supports state partitions that can work in parallel if no dependencies exist.

3. **Confidentiality.** The state of a channel remains internal to the channel peers. The only (if any) state revealed for cross-channel transactions should be necessary to verify that a transaction is valid (*e.g.,* does not create new assets).

8.3 Asset Management in a Single Channel

In this section, we describe a simple asset-management system on top of the *Unspent Transaction-Output* model (henceforth referred to as UTXO) that utilizes a single, non-confidential channel. In particular, we focus on the UTXO-based data model [191], as it is the most adopted data model in cryptocurrencies, for its simplicity and parallelizability.

8.3.1 Assets in Transactions

In a UTXO system, transactions are the means through which one or more *virtual* assets are managed. More specifically, *mint* transactions signify the introduction of new assets in the system and *spend* transactions signify the change of ownership of an asset that already exists in the system. If an asset is *divisible*, i.e., can be split into two or more assets of measurable

value, then a *spend* transaction can signify such a split, indicating the owners of each resulting component of the original asset.

Assets are represented in the transactions by transaction *inputs* and *outputs*. More specifically, in the typical UTXO model, an *input* represents the asset that is to be spent and an *output* represents the new asset that is created in response of the input assets' consumption. We can think of inputs and outputs representing different phases of the state of the same asset, where state includes its ownership (shares). Clearly, an input can be used only once, as after being spent, the original asset is substituted by the output assets, and stops being considered in the system. To ensure the single-spending of any given input, transactions are equipped with information authenticating the transaction creators as the owners of the (parts of the) assets that are referenced by the transaction inputs.

In more technical terms in the standard UTXO model, *input* fields implicitly or explicitly reference *output* fields of other transactions that have not yet been spent. At validation time, verifiers would need to ensure that the outputs referenced by the inputs of the transaction have not been spent; and upon transaction-commitment deem them as spent. To look up the status of each output at validation time efficiently, the UTXO model is equipped with a pool of *unspent transaction outputs* (UTXO pool).

8.3.2 UTXO Pool

The UTXO pool is the list of transaction outputs that have not yet been *spent*. We say that an output is *spent* if a transaction that references it in its inputs is included in the list of ledger's valid transactions.

To validate a transaction, peers check if (1) the transaction inputs refer to outputs that appear in the UTXO pool as well as (2) that the transaction's creators own these outputs. Other checks take place during the transaction validation, *i.e.,* input-output consistency checks. After these checks are successfully completed, the peers mark the outputs matching the transaction's inputs as spent and add to the pool the freshly created outputs. Hence, the pool consistently includes "unspent" outputs.

8.3.3 Asset or Output Definition

An *asset* is a logical entity that sits behind transaction outputs, implicitly referenced by transaction outputs. As such the terms output and asset can be used interchangeably. An output (the corresponding asset) is described by the following fields:

- *namespace*, the namespace the output belongs to (*e.g.,* a channel);
- *owner*, the owner of the output
- *value*, the value of the asset the output represents (if divisible);

- *type*, the type of the asset the output represents (if multiple types exist).

Depending on the privacy requirements and properties of the ledger they reside, outputs provide this information in the clear (*e.g.,* Bitcoin [191] outputs) or in a concealed form (*e.g.,* ZeroCoin [182], ZeroCash [218]). Privacy-preserving outputs are required to be cryptographically bound to the value of each of the fields describing them, whereas its plaintext information should be available to the owner of the output.

8.3.4 UTXO operations

We elaborate on the UTXO system functions where we adopt the following notation. For a sequence of values $x_1, \ldots, x_i$, we use the notation $[x_i] = (x_1, \ldots, x_i)$. By slight abuse of notation, we write $x_1 = [x_1]$. We denote algorithms by sans-serif fonts. Executing an algorithm algo on input y is denoted as $y \leftarrow \mathsf{algo}(x)$, where y can take on the special value $\perp$ to indicate an error.

A UTXO system exposes the following functions:

- $\langle \mathscr{U}, \text{pool} \rangle \leftarrow \mathsf{Setup}(\kappa)$ that enables each user to issue one or more identities by using security parameter κ. Henceforth, we denote by sec_{user} the secret information associated to a user with identity user. Setup also generates privileged identities, i.e., identities allowed to mint assets to the system, denoted as adm. Finally, Setup initializes the pool pool to $\emptyset$ and returns the set of users in the system $\mathscr{U}$ and pool.
- $\langle \text{out}, \text{sec}_{\text{out}} \rangle \leftarrow \mathsf{ComputeOutput}(\text{nspace}, \text{owner}, \text{value}, \text{type})$, to obtain an output representing the asset state as reflected in the function's parameters. That is, the algorithm would produce an output that is bound to namespace nspace, owned by owner, and represents an asset of type type, and value value. As mentioned before, depending on the nature of the system the result of the function could output two output components, one that is to be posted on the ledger as part of a transaction (out) and a private part to be maintained at its owner side (sec_{out}).
- $\text{ain} \leftarrow \mathsf{ComputeInput}(\text{out}, \text{sec}_{\text{out}}, \text{pool})$, where, on input an asset pool pool, an output out, and its respective secrets, the algorithm returns a representation of the asset that can be used as transaction input ain. In Bitcoin, an input of an output is a direct reference to the latter, i.e., it is constructed to be the hash of the transaction where the output appeared in the ledger, together with the index of the output. In ZeroCash, an input is constructed as a combination of a serial number and a zero-knowledge proof that the serial corresponds to an unspent output of the ledger.
- $\text{tx} \leftarrow \mathsf{CreateTx}([\text{sec}_{\text{owner}_i}], [\text{ain}_i], [\text{out}_j])$, that creates a transaction tx to request the consummation of inputs $\{\text{ain}_k\}_{k=1}^{i}$ into outputs $\{\text{out}_k\}_{k=1}^{j}$. The function takes also as input the secrets of the owners of the outputs referenced by the inputs and returns tx. Notice that the same function can be used to construct *mint* transactions, where the input gives its place to the freshly introduced assets description.

- pool′ ← ValidateTx(nspace, tx, pool), that validates transaction inputs w.r.t. pool pool, and their consistency with transaction outputs and namespace nspace. It subsequently updates the pool with the new outputs and spent inputs and returns its new version pool′. Input owner of mint transactions is the administrators adm.

Properties Regardless of its implementation, an asset management system should satisfy the properties defined below:

- *Validity*: Let tx be a transaction generated from a valid input ain according to some pool pool, i.e., generated via a successful call to tx ← CreateTx($\text{sec}_{\text{owner}}$, ain, out′), where ain ← ComputeInput(out, sec_{out}, pool), owner is the owner of out′, and out′ ∉ pool. Validity requires that a call to pool′ ← ValidateTx(tx, pool) succeeds, i.e. pool′ ≠ ⊥, and that pool′ = (pool \ {out}) ∪ {out′}.
- *Termination*: Any call to the functions exposed by the system eventually return the expected return value or ⊥.
- *Unforgeability*. Let an output out ∈ pool with corresponding secret sec_{out} and owner secret $\text{sec}_{\text{owner}}$ that is part of the UTXO pool pool; unforgeability requires that it is computationally hard for an attacker without sec_{out} and $\text{sec}_{\text{owner}}$ to create a transaction tx such that ValidateTx(nspace, tx, pool) will not return ⊥, and that would mark out as spent.
- *Namespace consistency*. Let an output corresponding to a namespace nspace of a user owner. Namespace consistency requires that the adversary cannot compute any transaction tx referencing this output, and succeed in ValidateTx(nspace′, tx, pool), where nspace′ ≠ nspace.
- *Balance*. Let a user owner owning a set of unspent outputs $[\text{out}_i] \in$ pool. Let the collected value of these outputs for each asset type τ be value_τ. Balance property requires that owner cannot spend outputs of value more than value_τ for any asset type τ, assuming that it is not the recipient of outputs in the meantime, or colludes with other users owning more outputs. Essentially, it cannot construct a set of transactions $[\text{tx}_i]$ that are all accepted when sequentially[2] invoking ValidateTx(tx, pool) with the most recent versions of the pool pool, such that owner does not appear as the recipient of assets after the acquisition of $[\text{out}_i]$, and the overall spent value of its after that point exceeds for some asset type τ value_τ.

8.3.5 Protocol

We defined an asset output as, out = $\langle nm, o, t, v \rangle$, where nm is a namespace of the asset, o is the identity of its owner, t the type of the asset, and v its value. In its simplest implementation,

[2]This is a reasonable assumption, given we are referring to transactions appearing on a ledger.

the UTXO pool would be implemented as the list of available outputs and inputs would directly reference the outputs in the pool, *e.g.,* using its hash[3]. Clearly, a valid transaction for out's spending would require a signature with sec_o.

Asset Management in a single channel We assume two users Alice and Bob, with respective identities $\langle A, \text{sec}_A \rangle$ and $\langle B, \text{sec}_B \rangle$. There is only one channel ch in the system with a namespace ns_{ch} associated with ch, where both users have permission to access. We also assume that there are system administrators with secrets sec_{adm} allowed to mint assets in the system, and that these administrators are known to everyone.

Asset Management Initialization This requires the setup of the identities of the system administrators[4]. For simplicity, we assume there is one asset management administrator, $\langle \text{adm}, \text{sec}_{adm} \rangle$. The pool is initialized to include no assets, i.e., $\mathsf{pool}_{\mathsf{ch}} \leftarrow \emptyset$.

Asset Import The administrator creates a transaction tx_{imp}, as:

$$tx_{imp} \leftarrow \langle \emptyset, [\text{out}_n], \sigma \rangle,$$

where $\text{out}_k \leftarrow \mathsf{ComputeOutput}(ns_{ch}, \text{u}_\text{k}, t_k, v_k)$, (t_i, v_i) the type and value of the output asset out_k, u_k its owner and σ a signature on transaction data using sk_{adm}. Validation of tx_{imp} would result into $\mathsf{pool}_{\mathsf{ch}} \leftarrow \{\mathsf{pool}_{\mathsf{ch}} \cup \{[\text{out}_n]\}\}$.

Transfer of Asset Ownership Let $\text{out}_A \in \mathsf{pool}_{\mathsf{ch}}$ be an output owned by Alice, corresponding a description $\langle ns_{ch}, A, t, v \rangle$). For Alice to move ownership of this asset to Bob, it would create a transaction

$$\text{tx}_{move} \leftarrow \mathsf{CreateTx}(\text{sec}_A; \text{ain}_A, \text{out}_B),$$

where ain_A is a reference of out_A in $\mathsf{pool}_{\mathsf{ch}}$, and $\text{out}_B \leftarrow \mathsf{ComputeOutput}(ns_{ch}, B, t, v)$, the updated version of the asset, owned by Bob. tx_{move} has the form of $\langle \text{ain}_A, \text{out}_B, \sigma_A \rangle$ is a signature matching A. At validation of tx_{move}, $\mathsf{pool}_{\mathsf{ch}}$ is updated to no longer consider out_A as unspent, and include the freshly created output out_B:

$$\mathsf{pool}_{\mathsf{ch}} \leftarrow (\mathsf{pool}_{\mathsf{ch}} \setminus \{\text{out}_A\}) \cup \{\text{out}_B\}\,.$$

Discussion The protocol introduced above does provide a "secure" (under the security properties described above) asset management application within a single channel. More specifically, the *Validity* property follows directly from the correctness of the application where a transaction generated by using a valid input representation will be successfully validated by the peers after it is included in an ordered block. The *Unforgeability* is guaranteed from

[3] Different approaches would need to be adopted in cases where unlinkability between outputs and respective inputs is required.

[4] Can be a list of identities, or policies, or mapping between either of the two and types of assets.

the requirement of a valid signature corresponding to the owner of the consumed input when calling the ValidateTx function, and *Namespace consistency* is guaranteed as there is only one namespace in this setting. *Termination* follows from the liveness guarantees of the validating peers and the consensus run by orderers. Finally, *Balance* also follows from the serial execution of transactions that will spend the out the first time and return $\perp$ for all subsequent calls (there is no out in the pool).

The protocol can be extended to naively scale out. We can create more than one channel (each with its own namespace), where each one has a separate set of peers and each channel is unaware of the existence of other channels. Although each channel can have its own ordering service it has been shown [10], that the ordering service does not constitute a bottleneck. Hence, we assume that channels share the ordering service.

The naive approach has two shortcomings. First, assets cannot be transferred between channels, meaning that value is "locked" within a channel and is not free to flow wherever its owner wants. Second, the state of each channel is public as all transactions are communicated in plaintext to the orderers who act as a global passive adversary.

We deal with these problems by introducing (i) a step-wise approach on enabling cross-channel transactions depending on the functionality required and the underlying trust model (See, Section 8.4), and (ii) the notion of confidential channels (see Section 8.5). Further, for confidential channels to work, we adapt our algorithms to provide confidentiality while multiple confidential channels transact atomically.

8.4 Atomic Cross-Channel Transactions

In this section, we describe how we implement cross-channel transactions in permissioned blockchains (that enable the scale-out property as shown in Chapter 6). We introduce multiple protocols based on the functionality required and on the trust assumptions (that can be relaxed in a permissioned setting). First, in Section 8.4.1, we introduce a narrow functionality of 1-input-1-output transactions where Alice simply transfers an asset to Bob. Second, in Section 8.4.2, we extend this functionality to arbitrary transactions but assume the existence of a trusted channel among the participants. Finally, in Section 8.4.3, we lift this assumption and describe a protocol inspired by two-phase commit [248]. These protocols do not make timing assumptions but assume the correctness of the channels to guarantee fairness, unlike work in atomic cross-chain swaps.

Preliminaries We assume two users Alice (u_a), and Bob (u_b). We further assume that each channel has a validation policy and a set of oracles (as defined in Section 8.2). We assume that each channel is aware of the policies and the oracles that are authoritative over the asset-management systems in each of the rest of the channels.

Communication of pools content across channels. On a regular basis, each channel advertises its pool content to the rest of the channels. More specifically, the oracles of the asset management system in each channel are responsible for regularly advertise a commitment of the content of the channel's pool to the rest of the channels. Such commitments can be the full list of assets in the pool or, for efficiency reasons, the Merkle root of a deterministically ordered list of asset outputs created on that channel.

For the purpose of this simplistic example, we assume that for each channel $\mathsf{ch_i}$, a commitment (*e.g.,* Merkle root) of its pool content is advertised to all the other channels. That is, each channel $\mathsf{ch_i}$ maintains a table with the following type of entries: $\langle \mathsf{ch_j}, cmt_j \rangle$, $j \neq i$, where cmt_j the commitment corresponding to the pool of channel with identifier $\mathsf{ch_j}$. We will refer to this pool by $\mathsf{pool_j}$.

8.4.1 Asset Transfer across Channels

Let out_A be an output included in the unspent output pool of $\mathsf{ch_1}$, $\mathsf{pool_1}$, corresponding to

$$\text{out}_A \leftarrow \mathsf{ComputeOutput}(\mathsf{ch_1}, u_a, t, v)$$

i.e., an asset owned by Alice, active on $\mathsf{ch_1}$. For Alice to move ownership of this asset to Bob and in channel with identifier $\mathsf{ch_2}$, she would first create a new asset for Bob in $\mathsf{ch_2}$ as

$$\text{out}_B \leftarrow \mathsf{ComputeOutput}(\mathsf{ch_2}, u_b, t, v)$$

she would then create a transaction

$$\text{tx}_{\text{move}} \leftarrow \mathsf{CreateTx}(\text{sec}_A; \text{ain}_A, \text{out}_B),$$

where ain_A is a reference of out_A in $\mathsf{pool_1}$. Finally, sec_A is a signature matching pk_A, and ownership transfer data.

At validation of tx_{move}, it is first ensured that $\text{out}_A \in \mathsf{pool_1}$, and that $\text{out}_A.namespace = \mathsf{ch_1}$. out_A is then removed from $\mathsf{pool_1}$ and out_B is added to it, i.e.,

$$\mathsf{pool_1} \leftarrow (\mathsf{pool_1} \setminus \{\text{out}_A\}) \cup \{\text{out}_B\}\,.$$

Bob waits until the commitment of the current content of $\mathsf{pool_1}$ is announced. Let us call the latter $\mathsf{view_1}$. Then Bob can generate a transaction "virtually" spending the asset from $\mathsf{pool_1}$ and generating an asset in $\mathsf{pool_2}$. The full transaction will happen in $\mathsf{ch_2}$ as the spend asset's namespace is $\mathsf{ch_2}$. More specifically, Bob creates an input representation

$$\{\text{ain}_B\} \leftarrow \mathsf{ComputeInput}(\text{out}_B; \text{sec}_B, \pi_B)$$

of the asset out_B that Alice generated for him. Notice that instead of the pool, Bob needs to

provide π_B, we explain below why this is needed to guarantee the balance property. Finally, Bob generates a transaction using ain_B.

To be ensured that the out_B is a valid asset, Bob needs to be provided with a proof, say π_B, that an output matching its public key and ch_2 has entered pool_1, matching view_1. For example, if view_1 is the root of the Merkle tree of outputs in pool_1, π_B could be the sibling path of out_B in that tree with out_B. This proof can be communicated from the oracles of ch_1 to the oracles of ch_2 or be directly pulled by Bob and introduced to ch_2. Finally, in order to prevent Bob from using the same proof twice (*i.e.,* perform a replay attack) pool_2 need to be enhanced with a set of spent cross-transaction outputs (ScTXOs) that keep track of all the output representations out_X that have been already redeemed in another tx_{cross}. The out_B is extracted from π_B.

The *validity* property holds by extending the asset-management protocol of every channel to only accept transactions that spend assets that are part of the channel's name-space. *Unforgeability* holds as before, due to the requirement for Alice and Bob to sign their respective transactions. *Namespace Consistency* holds as before, as validators of each channel only validate consistent transactions; and *Termination* holds because of the liveness guarantees of ch_1 and ch_2 and the assumption that the gossiped commitments will eventually arrive at all the channels. Finally, the *Balance* property holds as Alice can only spend her asset once in ch_1, which will generate a new asset not controlled by Alice anymore. Similarly, Bob can only use his proof once as out_B will be added in the ScTXO list of pool_2 afterwards.

8.4.2 Cross-Channel Trade with a Trusted Channel

The approach described above works for cases where Alice is altruistic and wants to transfer an asset to Bob. However, more complicated protocols (*e.g.,* fair exchange) are not supported, as they need atomicity and abort procedures in place. For example, if Alice and Bob want to exchange an asset, Alice should be able to abort the protocol if Bob decides to not cooperate. With the current protocol, this is not possible as Alice assumes that Bob wants the protocol to finish and has nothing to win by misbehaving.

A simple approach to circumvent this problem is to assume a commonly trusted channel ch_t from all actors. This channel can either be an agreed upon "fair" channel or any of the channels of the participants, as long as all participants are able to access the channel and create/spend assets on/from it. The protocol uses the functionality of the asset transfer protocol described above (Section 8.4.1) to implement the Deposit and Withdraw subprotocols. In total, it exposes three functions and enables a cross-channel transaction with multiple inputs and outputs:

1. **Deposit:** All parties that contribute inputs transfer the assets to ch_t but maintain control over them by assigning the new asset in ch_t on their respective public keys.

2. **Transact:** When all input assets are created in ch_t, a tx_{cross} is generated and signed by all ain owners. This tx_{cross} has the full logic of the trade. For example, in the fair exchange,

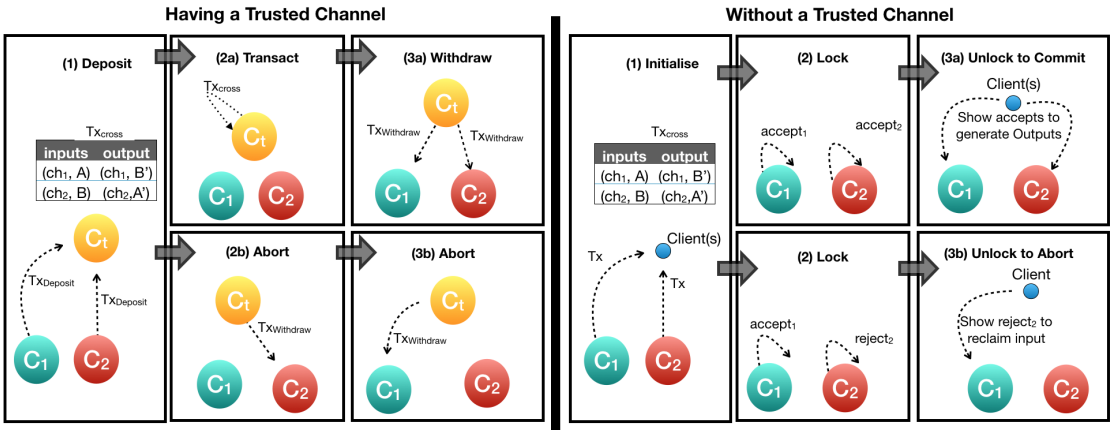

Figure 8.1 – Cross-channel transaction architecture overview with (8.4.2) and without (8.4.3) a trusted channel

it will have two inputs and two outputs. This tx_{cross} is validated as an atomic state update in ch_t.

3. **Withdraw:** Once the transaction is validated, each party that manages an output transfers their newly minted assets from ch_t to their respective channels ch_{o_i}

Any input party can decide to abort the protocol by transferring back the input asset to their channel, as they always remain in control of the asset.

The protocol builds on top of the asset-transfer protocol and inherits its security properties to the extent of the Deposit and Withdraw sub-protocols. Furthermore, the trusted channel is only trusted to provide the necessary liveness for assets to be moved across channels, but it cannot double-spent any asset as they still remain under the control of their rightful owners (bound to the owner's public key). As a result, the asset-trade protocol satisfies the asset-management security requirements because it can be implemented by combining the protocol of Section 8.4.1 for the "Transact" function inside ch_t and the asset-transfer protocol of Section 8.4.2 for "Withdraw" and "Deposit".

8.4.3 Cross-Channel Trade without a Trusted Channel

A mutually trusted channel (as assumed above), where every party is permitted to generate and spend assets, might not always exist; in this section, we describe a protocol that lifts this assumption. The protocol is inspired by the Atomix protocol [150], but addresses implementation details that are ignored in Atomix, such as how to represent and communicate proofs, and it is more specialized to our asset management model.

1. **Initialize.** The transacting parties create a tx_{cross} whose inputs spend assets of some input channels (ICs) and whose outputs create new assets in some output channels

(OCs). More concretely.

If Alice wants to exchange $\mathsf{out_A}$ from $\mathsf{ch_1}$ with Bob's $\mathsf{out_B}$ from $\mathsf{ch_2}$. Alice and Bob work together to generate the $\mathrm{tx_{cross}}$ as

$$\mathrm{tx_{cross}} \leftarrow \mathsf{CreateTx}([\mathrm{sec}_A, \mathrm{sec}_B]; [\mathrm{ain}_A, \mathrm{ain}_B]; [\mathrm{out}_A, \mathrm{out}_B])$$

where ain_A, ain_B are the input representations that show the assets to exist in the respective pools.

2. **Lock.** All input channels internally spend the assets they manage and generate a new asset bound to the transaction (we call it the "locked asset"), by using a collision-resistant Hash function to derive the name-space of the new asset, as $H(\mathrm{tx_{cross}})$[5]. The locked asset's value is either equal to the sum of the assets previously spent on that channel or 0, depending on whether the $\mathrm{tx_{cross}}$ is valid according to the current state. In both cases, there is a new asset added in $\mathsf{pool_i}$. Or in our example:

   Alice submits $\mathrm{tx_{cross}}$ to $\mathsf{ch_2}$, which generates the "locked" asset for $\mathrm{tx_{cross}}$. Alice then receives π_B, which shows that out_B is locked for $\mathrm{tx_{cross}}$ and is represented by $\mathsf{out_{B'}}$, which is the locked asset that is generated specifically for $\mathrm{tx_{cross}}$ and is locked for Alice but not spendable by Alice. Specifically,

   $$asset_{2'} = \langle H(\mathrm{tx_{cross}}), t, v \rangle,$$

   where v is either equal to the value of asset_2 or 0, depending on whether asset_2 was already spent. Same process happens for Bob. Notice that the namespace of the asset change to $H(\mathrm{tx_{cross}})$ indicates that this asset can only be used as proof of existence and not spent again in $\mathsf{ch_2}$.

3. **Unlock.** Depending on the outcome of the lock phase, the clients are able to either commit or abort their transaction.

   (a) **Unlock to Commit.** If all ICs accepted the transaction (generated locked assets with non-zero values), then the respective transaction can be committed.

   Each holder of an output creates an unlock-to-commit transaction for his channel; it consists of the lock transaction and an oracle-generated proof for each input asset (*e.g.,* against the gossiped MTR). Or in our example:

   Alice (and Bob respectively) collects $\pi_{A'}$ and $\pi_{B'}$ which correspond to the proofs of existence of $\mathrm{out}_{A'}, \mathrm{out}_{B'}$ and submits in $\mathsf{ch_1}$ an unlock-to-commit transaction:

   $$\mathrm{tx_{uc}} \leftarrow \mathsf{CreateTx}([\pi_{A'}, \pi_{B'}]; [\mathrm{ain}_{1'}, \mathrm{ain}_{2'}]; [\mathrm{out}_{A''}];)$$

   The transaction is validated in $\mathsf{ch_1}$ creating a new asset ($\mathsf{out_{A''}}$), similar to the one Bob spent at $\mathsf{ch_2}$, as indicated by $\mathrm{tx_{cross}}$.

[5] The transaction's hash is an identifier for a virtual channel created only for this transaction

(b) **Unlock to Abort.** If, however, at least one IC rejects the transaction (due to a double-spent), then the transaction cannot be committed and has to abort. In order to reclaim the funds locked in the previous phase, the client must request the involved ICs that already spent their inputs, to re-issue these inputs. Alice can initiate this procedure by providing the proof that the transaction has failed in ch_2. Or in our case if Bob's asset validation failed, then there is an asset $\text{out}_{B'}$ with zero value and Alice received from ch_2 the respective proof $\pi'_{B'}$. Alice will then generate an unlock-to-abort transaction:

$$\text{tx}_{\text{ua}} \leftarrow \mathsf{CreateTx}([\pi_{B'}], [ain_{2'}]; [\text{out}_{A''}])$$

which will generate a new asset $\text{out}_{A''}$ that is identical to out_A and remains under the control of Alice

Security Arguments Under our assumptions, channels are collectively honest and do not fail hence propagate correct commitments of their pool (commitments valid against the validation policy).

Validity and *Namespace Consistency* hold because every channel manages its own namespace and faithfully executes transactions. *Unforgeability* holds as before, due to the requirement for Alice and Bob to sign their respective transactions and the tx_{cross}.

Termination holds if every tx_{cross} eventually commits or aborts, meaning that either a transaction will be fully committed or the locked funds can be reclaimed. Based on the fact that all channels always process all transactions, each IC eventually generates either a commit-asset or an abort-asset. Consequently, if a client has the required number of proofs (one per input), then the client either holds all commit-assets (allowing the transaction to be committed) or at least one abort-asset (forcing the transaction to abort), but as channels do not fail, the client will eventually hold enough proof. Termination is bound to the assumption that some client will be willing to initiate the unlock step, otherwise, his assets will remain unspendable. We argue that failure to do such only results in harm of the asset-holder and does not interfere with the correctness of the asset-management application.

Finally, *Balance* holds as cross-channel transactions are atomic and are assigned to specific channels who are solely responsible for the assets they control (as described by validity) and generate exactly one asset. Specifically, if all input channels issue an asset with value, then every output channel unlocks to commit; if even one input channel issues an asset with zero value, then all input channels unlock to abort; and if even one input shard issues an asset with zero value, then no output channel unlocks to commit. As a result, the assigned channels do not process a transaction twice and no channel attempts to unlock without a valid proof.

8.5 Using Channels for Confidentiality

So far we have focused on enabling transactions between channels that guarantee fairness among participants. This means that no honest participant will be worse off by participating in one of the protocols. Here, we focus on providing confidentiality among the peers of a channel, assuming that the orderers upon which the channel relies for maintaining the blockchain are not fully trusted hence might leak data.

Strawman Solution We start with a simple solution that can be implemented with vanilla channels [10]. We define a random key k and a symmetric encryption algorithm that is sent in a private message to every participating peer. All transactions and endorsements are encrypted under k then sent for ordering, hence the confidentiality of the channel is protected by the unpredictability of the symmetric encryption algorithm.

This strawman protocol provides the confidentiality we expect from a channel, but its security is static. Even though peers are trusted for confidentiality, all it takes for an adversary to compromise the full past and future confidential transactions of the system is to compromise a single peer and recover k. Afterwards the adversary can collude with a Byzantine order to use the channels blockchain as a log of the past and decrypt every transaction, as well as keep receiving future transactions from the colluding orderer.

8.5.1 Deploying Group Key Agreement

To work around the attack, we first need to minimize the attack surface. To achieve this we need to think of the peers of a channel, as participants of a confidential communication channel and provide similar guarantees. Specifically, we guarantee the following properties.

1. **Forward Secrecy:** A passive adversary that knows one or more old encryption keys k_i, cannot discover any future encryption key k_j where $i < j$
2. **Backward Secrecy:** A passive adversary that knows one or more encryption keys k_i, cannot discover any previous encryption key k_j where $j < i$
3. **Group Key Secrecy:** It is computationally infeasible for an adversary to guess any group key k_i
4. **Key Agreement:** For an epoch i all group members agree on the epoch key k_i

There are two types of group key agreement we look into:

Centralized group-key distribution: In these systems, there is a dedicated server that sends the symmetric key to all the participants. The centralized nature of the key creation is scalable, but might not be acceptable even in a permissioned setting where different organizations participating in a channel are mutually suspicious.

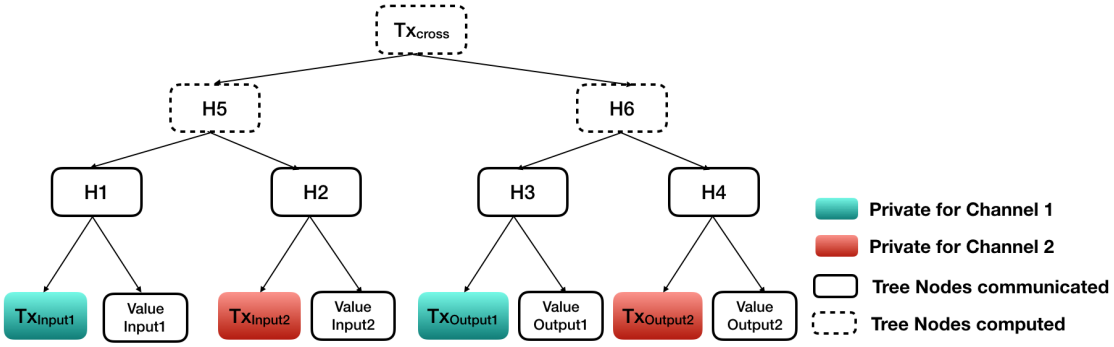

Figure 8.2 – Privacy-Preserving Cross-Channel Transaction structure

Contributory group-key management: In these systems, each group member contributes a share to the common group key, which is then computed by each member autonomously. These protocols are a natural fit to decentralized systems such as distributed ledgers, but they scale poorly.

We use the existence of the validation policy as an indication of the trusted entities of the channel (*i.e.,* the oracles) and create a more suitable protocol to the permissioned setting. Another approach could be to introduce a key-management policy that defines the key-generation and update rules but, for simplicity, we merge it with the validation policy that the peers trust anyway. We start with a scalable contributory group-key agreement protocol [146], namely the Tree-Based Group Diffie-Hellman system. However, instead of deploying it among the peers as contributors (which would require running view-synchronization protocols among them), we deploy it among the smaller set of oracles of the channel. The oracles generate symmetric keys in a decentralized way, and the peers simply contact their favorite oracle to receive the latest key. If an oracle replies to a peer with an invalid key, the peer can detect it because he can no longer decrypt the data, hence he can (a) provably blame the oracle and (b) request the key from another oracle.

More specifically, we only deploy the group-join and group-leave protocols of Kim et al. [146], because we do not want to allow for splitting of the network, which might cause forks on the blockchain. We also deploy a group-key refresh protocol that is similar to group-leave, but no oracle is actually leaving.

8.5.2 Enabling Cross-Shard Transactions among Confidential Channels

In the protocols we mentioned in Section 8.4, every party has full visibility on the inputs and outputs and is able to link the transfer of coins. However, this might not be desirable. In this section, we describe a way to preserve privacy during cross-channel transactions within each asset's channel.

For example, we can assume the existence of two banks, each with its own channel. It would

be desirable to not expose intra-channel transactions or account information when two banks perform an interbank asset-transfer. More concretely, we assume that Alice and Bob want to perform a fair exchange. They have already exchanged the type of assets and the values they expect to receive. The protocol can be extended to store any kind of ZK Proofs the underlying system supports, as long as the transaction can be publicly verified based on the proofs.

To provide the obfuscation functionality, we use Merkle trees. More specifically, we represent a cross-shard transaction as a Merkle tree (see Figure 8.2), where the left branch has all the inputs lexicographically ordered and the right branch has all the outputs. Each input/output is represented as a tree node with two leaves: a private leaf with all the information available for the channel and a public leaf with the necessary information for third-party verification of the transaction's validity.

The protocol for Alice works as follows:

Transaction Generation:

1. Input Merkle-Node Generation: Alice generates an input as before and a separate Merkle leaf that only has the type of the asset and the value. These two leaves are then hashed together to generate their input Merkle node.

2. Output Merkle-Node Generation: Similarly, Alice generates an Output Merkle node, that consists of the actual output (including the output address) on the private leaf and only the type and value of the asset expected to be credited on the public.

3. Transaction Generation: Alice and Bob exchange their public Input and Output Merkle-tree nodes and autonomously generate the full Merkle tree of the transaction.

Transaction Validation:

1. Signature Creation: Alice signs the MTR of the tx_{cross}, together with a bitmap of which leaves she has seen and accepts. Then she receives a similar signature from Bob and verifies it. Then Alice hashes both signatures and attaches them to the full transaction. This is the tx_{cross} that she submits in her channel for validation. Furthermore, she provides her full signature, which is logged in the channel's confidential chain but does not appear in the pool; in the pool the generated asset is $H(\text{tx}_{\text{cross}})$.

2. Validation: Each channel validates the signed transaction (from all inputs inside the channel's state) making sure that the transaction is semantically correct (*e.g.,* does not create new assets). They also check that the publicly exposed leaf of every input is well generated (*e.g.,* value and type much). Then they generate the new asset ($H(\text{tx}_{\text{cross}})$ as before) that is used to provide proof-of-commitment/abortion. The rest of the protocol (*e.g.,* Unlock phase) is the same as Section 8.4.3.

Security & Privacy Arguments The atomicity of the protocol is already detailed above. Privacy is achieved because the source and destination addresses (accounts) are never exposed outside the shard and the signatures that authenticate the inputs inside the channel are only exposed within the channel. We also describe the security of the system outside the atomic commit protocol. More specifically,

1. Every tx_{cross} is publicly verifiable to make sure that the net-flow is zero, either by exposing the input and output values or by correctly generating ZK-proofs.
2. The correspondence of the public and the private leaf of a transaction is fully validated by the input and/or output channel, making sure that its state remains correct.
3. The hash of the tx_{cross} is added in the pool to represent the asset. Given the collision resistance of a hash function, this signals to all other channels that the private leaves correspond to the transaction have been seen, validated and accepted.

The scheme can be further enhanced to hide the values using Pedersen commitments [201] and range proofs similar to confidential transactions [204]. In such an implementation the Pedersen commitments should also be opened on the private leaf for the consistency checks to be correctly done.

8.6 Case Study: Cross-Shard Transactions on Hyperledger Fabric

In order to implement the cross-channel support on Fabric v1.1, we start with the current implementation of FabCoin [10] that implements an asset-management protocol similar to the one introduced in Section 8.3.5.

Channel-Based Implementation

As described by Androulaki et al. [10], a Fabric network can support multiple blockchains connected to the same ordering service. Each such blockchain is called a *channel*. Each channel has its own configuration that includes all the functioning metadata, such as defining the membership service providers that authenticate the peers, how to reach the ordering service, and the rules to update the configuration itself. The genesis block of a channel contains the initial configuration. The configuration can be updated by submitting a *reconfiguration transaction*. If this transaction is valid with the respect to the rules described by the current configuration, then it gets committed in a block containing only the reconfiguration transaction, and the chances are applied.

In this work, we extend the channel configuration to include the metadata to support cross-channel transactions. Specifically, the configuration lists the channels with which interaction is allowed; we call them *friend channels*. Each entry also has a *state-update validation policy*,

to validate the channel's state-updates, the identities of the oracles of that channel, that will advertise state-update transactions, and the current commitment to the state of that channel. The configuration block is also used as a *lock-step* that signals the view-synchrony needed for the oracles to produce the symmetric-key of the channel. If an oracle misbehaves, then a new configuration block will be issued to ban it.

Finally, we introduce a new entity called *timestamper* (inspired by recent work in software updates [195]) to defend against freeze attacks where the adversary presents a stale configuration block that has an obsolete validation policy, making the network accepting an incorrect state update. The last valid configuration is signed by the timestampers every *interval*, defined in the configuration, and (assuming loosely synchronized clocks) guarantees the freshness of state updates[6].

Extending FabCoin to Scale-out

In FabCoin [10] each asset is represented by its current output state that is a tuple of the form (*txid.j*,(*value*, *owner*, *type*)). This representation denotes the the asset created as the j-th output of a transaction with identifier *txid* that has *value* units of asset *type*. The output is owned by the public key denoted as *owner*.

To support cross-channel transactions, we extend FabCoin transactions by adding one more field, called *namespace*, that defines the channel that manages the asset
(*i.e.,* (*txid.j*,(*namespace*, *value*, *owner*, *type*))).

Periodically, every channel generates a *state commitment* to its state, this can be done by one or more channel's oracles. This state commitment consists of two components: (i) the root of the Merkle tree built on top of the UTXO pool, (ii) the hash of the current configuration block with the latest timestamp, which is necessary to avoid freeze attacks.

Then, the oracles of that channel announce the new state commitment to the friend channels by submitting specific transactions targeting each of these friend channels. The transaction is committed if (i) the hashed configuration block is equal to the last seen configuration block, (ii) the timestamp is not "too" stale (for some time value that is defined per channel) and (iii) the transaction verifies against the state-updates validation policy. If those conditions hold, then the channel's configuration is updated with the new state commitment. If the first condition does not hold, then the channel is stale regarding the external channel it transacts with and needs to update its view.

Using the above state update mechanism, Alice and Bob can now produce verifiable proofs that certain outputs belong to the UTXO pool of a certain channel; these proofs are communicated to the interested parties differently, depending on the protocol. On the simple asset-transfer case (Section 8.4.1), we assume that Alice is altruistic (as she donates an asset to

[6] Unless both the timestamp role and the validation policy are compromised.

Table 8.1 – Atomic Commit Protocol on Fabric Channels

Protocol	Atomicity	Trust Assumption	Generality of Transactions	Privacy
Asset Transfer (Section 8.4.1)	Yes	Nothing Extra	1-Input-1-Ouptut	No
Trusted Channel (Section 8.4.2)	Yes	Trusted Intermediary Channel	N-Input-M-output	No
Atomic Commit (Section 8.4.3)	Yes	Nothing Extra	N-Input-M-output	No
Obfuscated Transaction AC (Section 8.5.2)	Yes	Nothing Extra	N-Input-M-output	Yes

Bob) and request the proofs from her channel that is then communicated off-band to Bob. On the asset trade with trusted channels (Section 8.4.2) Alice and Bob can independently produce the proofs from their channels or the trusted channel as they have visibility and access rights. Finally on the asset trade of Section 8.4.3, Alice and Bob use the signed cross-channel transaction as proof-of-access right to the channels of the input assets in order to obtain the proofs. This is permitted because the tx_{cross} is signed by some party that has access rights to the channel and the channels peers can directly retrieve the proofs as the asset's ID is derived from $H(tx_{cross})$.

8.7 Conclusion

In this chapter, we have redefined channels, provided an implementation guideline on Fabric [10] and formalized an asset management system. A channel is the same as a shard that has been already defined in previous work [74, 150]. Our first contribution is to explore the design space of sharding on permissioned blockchains where different trust assumptions can be made. We have introduced three different protocols that achieve different properties as described in Table 8.1. Afterwards, we have introduced the idea that a channel in a permissioned distributed ledger can be used as a confidentiality boundary and describe how to achieve this. Finally, we have merged the contributions to achieving a confidentiality preserving, scale-out asset management system, by introducing obfuscated transaction trees.

Part IV
Related Work and Concluding Remarks

9 Related Work

In this section, we summarize work that is related to the systems this thesis describes. We cover the topics of scaling blockchains, consensus-group membership management, generating good distributed randomness, enabling confidentiality of blockchains, and decentralized management of identities and certificates.

9.1 Scaling Blockchains

How to handle Bitcoin's performance shortcomings is one of the major controversies discussed in the community. But even though it is an urgent concern, so far no solution was found that all involved parties were satisfied with. A simple approach would be to increase the block size and with that the maximum throughput of Bitcoin. However, this might also negatively affect the security of the system and does not address the core of the problem: Bitcoin's probabilistic consistency.

ByzCoin and Bitcoin [191] share a similar objective: implement a state machine replication (SMR) system with open membership [42]. Both, therefore, differ from more classic approaches to Byzantine fault-tolerant SMRs with static or slowly changing consensus groups such as PBFT [55], Tendermint [46], or Hyperledger Fabric [10].

Bitcoin has well-known performance shortcomings; there are several proposals [167, 250] on how to address these. The GHOST protocol [233] changes the chain selection rule when a fork occurs. While Bitcoin declares the fork with the most proof-of-work as the new valid chain, GHOST instead chooses the entire subtree that received the most computational effort. Put differently, the subtree that was considered correct for the longest time will have a high possibility of being selected, making an intentional fork much harder. One limitation of GHOST is that no node will know the full tree, as invalid blocks are not propagated. While all blocks could be propagated, this makes the system vulnerable to DoS attacks since an adversary can simply flood the network with low-difficulty blocks.

Off-chain transactions, an idea that originated from the two-point channel protocol [127], are

another alternative to improve latency and throughput of the Bitcoin network. Other similar proposals include the Bitcoin Lightning Network [205] and micro-payment channels [78], which allow transactions without a trusted middleman. They use contracts so that any party can generate proof-of-fraud on the main blockchain and thereby deny revenue to an attacker. Although these systems enable faster cryptocurrencies, they do not address the core problem of scaling SMR systems, thus sacrificing the open and distributed nature of Bitcoin. Finally, the idea behind sidechains [17] is to connect multiple chains with each other and enable the transfer of Bitcoins from one chain to another. This enables the workload distribution to multiple subsets of nodes that run the same protocol.

There are several proposals that, like BYZCOIN, target the consensus mechanism and try to improve different aspects. Ripple [224] implements and runs a variant of PBFT that is low-latency and based on collectively-trusted subnetworks with slow membership changes. The degree of decentralization depends on the concrete configuration of an instance. Tendermint [46] also implements a PBFT-like algorithm, but evaluates it with at most 64 "validators". Furthermore, Tendermint does not address important challenges such as the link-bandwidth between validators, which we found to be a primary bottleneck. PeerCensus [77] and Hybrid Consensus [198] are interesting alternative that shares similarities with BYZCOIN, but are only of theoretical interest.

Finally, Stellar [176] proposes a novel consensus protocol named Federated Byzantine Agreement, which introduces Quorum slices that enable a BFT protocol "open for anyone to participate". Its security, however, depends on a nontrivial and unfamiliar trust model requiring correct configuration of trustees by each client.

9.2 Comparison of OMNILEDGER with Prior Work

Table 9.1 – Comparison of Distributed Ledger Systems

System	Scale-Out	Cross-Shard Atomicity	State Blocks	Measured Scalability (# of Validators)	Estimated Time to Fail	Measured Latency
RSCoin [74]	In Permissioned	Partial	No	30	N/A	1 sec
Elastico [173]	In PoW	No	No	1600	1 hour	800 sec
ByzCoin [149]	No	N/A	No	1008	19 years	40 sec
Bitcoin-NG [95]	No	N/A	No	1000	N/A	600 sec
PBFT [46, 10]	No	N/A	No	16	N/A	1 sec
Nakamoto [191]	No	N/A	No	4000	N/A	600 sec
OMNILEDGER	Yes	Yes	Yes	2400	68.5 years	1.5 sec

The growing interests in scaling blockchains have produced a number of prominent systems that we compare in Table 9.1. ByzCoin [149] is a first step to scalable BFT consensus, but cannot scale out. Elastico is the first open scale-out DL, however, it suffers from performance and security challenges that we have already discussed in Section 6.2.2. RSCoin [74] proposes sharding as a scalable approach for centrally banked cryptocurrencies. RSCoin relies on a trusted source of randomness for sharding and auditing, making its usage problematic in

trustless settings. Furthermore, to validate transactions, each shard has to coordinate with the client and instead of running BFT, RSCoin uses a simple two-phase commit, assuming that safety is preserved if the majority of validators is honest. This approach, however, does not protect from double spending attempts by a malicious client colluding with a validator.

In short, prior solutions [74, 149, 173] achieve only two out of the three desired properties; decentralization, long-term security, and scale-out, as illustrated in Figure 6.1. OMNILEDGER overcomes this issue by scaling out, as far as throughput is concerned, and by maintaining consistency to the level required for safety, without imposing a total order.

Bitcoin-NG scales Bitcoin without changing the consensus algorithm by observing that the PoW process does not have to be the same as the transaction validation process; this results in two separate timelines: one slow for PoW and one fast for transaction validation. Although Bitcoin-NG significantly increases the throughput of Bitcoin, it is still susceptible to the same attacks as Bitcoin [113, 13].

9.3 Consensus Group Membership and Stake

Unlike permissioned blockchains [74], where the validators are known, permissionless blockchains need to deal with the potential of Sybil attacks [84] to remain secure. Bitcoin [191] suggested the use of Proof-of-Work (PoW), where validators (aka miners) create a valid block by performing an expensive computation (iterating through a nonce and trying to brute-force a hash of a block's header such that it has a certain number of leading zeros). Bitcoin-NG [95] uses this PoW technique to enable a Sybil-resistant generation of identities. There are certain issues associated with PoW, such as the waste of electricity [79] and the fact that it causes recentralization [134] to mining pools. Other approaches for establishing Sybil-resistant identities such as Proof-of-Stake (PoS) [116], Proof-of-Burn (PoB) [247] or Proof-of-Personhood [43] overcome PoW's problems and are promising ways toward ecological blockchain technology.

9.4 Randomness Generation and Beacons

Generation of public randomness has been studied in various contexts. In 1981, Blum proposed the first coin flipping protocol [36]. Rabin introduced the notion of cryptographic randomness beacons in 1983 [210]. NIST later launched such a beacon to generate randomness from high-entropy sources [193]. Centralized randomness servers have seen limited adoption, however, in part because users must rely on the trustworthiness of the party that runs the service.

Other approaches attempt to avoid trusted parties [206, 41, 18, 60]. Bonneau et al. [41] use Bitcoin to collect entropy, focusing on analyzing the financial cost of a given amount of bias rather than preventing bias outright. Lenstra et al. [165] propose a new cryptographic primitive,

a *slow hash*, to prevent a client from biasing the output. This approach is promising but relies on new and untested cryptographic hardness assumptions, and assumes that everyone observes the commitment before the slow hash produces its output. If an adversary can delay the commitment messages and/or accelerate the slow hash sufficiently, he can see the hash function's output before committing, leaving the difficult question of how slow is "slow enough" in practice. Other approaches use lotteries [18], or financial data [60] as public randomness sources.

An important observation by Gennaro et al. [111] is that in many distributed key generation protocols [202] an attacker can observe public values of honest participants, use this knowledge to influence the protocol run, and disqualify honest peers hence would be able to bias output. To mitigate this attack, the authors propose to delay the disclosure of the protocol's public values after a "point-of-no-return" at which point the attacker cannot influence the output anymore, as it has been already fixed before and, in particular, the honest participants can finish the protocol without the cooperation of the attacker. We also use the concept of a "point-of-no-return" in our randomness generation protocols to prevent an adversary from biasing the output. However, their assumption of a fully synchronous network is unrealistic for real-world scenarios. Cachin et al., propose an asynchronous distributed coin tossing scheme for public randomness generation [49], which relies on a trusted setup dealer. We improve on that by letting multiple nodes deal secrets and combine them for randomness generation in our protocols. Finally, Kate et al. [141], introduced an approach to solving distributed key-generation in large-scale asynchronous networks, such as the Internet. The communication complexity of their solution, similar to Gennaro's and Cachin's prevents scalability to large numbers of nodes. Our protocols use sharding to limit communication overheads to linear increases, which enables RANDHOUND and RANDHERD to scale to hundreds of nodes.

Applications of public randomness are manifold and include the protection of hidden services in the Tor network [119], selection of elliptic curve parameters [18, 165], Byzantine consensus [196], electronic voting [4], random sharding of nodes into groups [121], and non-interactive client-puzzles [124]. In all of these cases, both RANDHOUND and RANDHERD may be useful for generating bias-resistant, third-party verifiable randomness. For example, RANDHOUND could be integrated into the Tor consensus mechanism to help the directory authorities generate their daily random values in order to protect hidden services against DoS or popularity estimation attacks.

9.5 Confidential Blockchains

CALYPSO is based on threshold cryptography, first used to replicate services [213]. Our approach can also be considered a version of proxy re-encryption [34], where the proxy is actually a decentralized service that can withstand individual malicious proxies. The decentralized data management platform Enigma [256, 257] provides comparable functionality to CALYPSO. Users own and control their data and a blockchain enforces access control by logging valid

requests (as per the on-chain policy). However, Enigma stores the confidential data at a non-decentralized storage provider who can read and/or decrypt the data or refuse to serve the data even if there is a valid on-chain proof. The storage provider in Enigma is, therefore, a single point of compromise/failure.

Other projects [15, 86, 133, 226, 257] commonly rely on centralized key-management and/or storage systems as well and hence suffer from comparable issues with respect to atomicity and robustness against malicious service providers. Vanish [108] is another secure data-sharing system which ensures that no-longer-usable data self-destructs to protect against accidental leakage. CALYPSO can provide similar functionality by adding time-outs to write transactions after which (honest) trustees destroy their secret key shares making the secret inaccessible. Vanish, however, relies on DHTs and is thus not as robust as the blockchain-based CALYPSO. Other privacy-focused blockchains [182, 218] do not address the issue of data sharing and access control but instead, focus on hiding identity and transaction data through zero-knowledge proofs.

9.6 Decentralized Identity & Certificate Management

Existing decentralized identity management systems, such as UIA [104] or SDSI/SPKI [214] enable users to control their identities but they lack authenticated updates via trust-delegating forward links of skipchains which enable SKIPPER to support secure long-term relationships between user identities and secure access control over shared data. OAuth2 [126] is an access-control framework where an authorization server can issue access tokens to authenticated clients which the latter can use to retrieve the requested data from a resource server. CALYPSO can emulate OAuth2 without any single points of compromise/failure where the access-control blockchain and the secret-management cothority act as decentralized versions of the authorization and resource servers, respectively. Further, thanks to CALYPSO's serialization of access requests and SIAM updates, it is not vulnerable to attacks exploiting race conditions when revoking access rights or access keys like OAuth2 [172]. ClaimChain [156] is a decentralized PKI where users maintain repositories of claims about their own and contacts' public keys. However, it permits transfer of access-control tokens, which can result in unauthorized access to the claims. Finally, Blockstack [7] uses Bitcoin to provide naming and identity functionality, but it does not support private-data sharing with access control. CALYPSO can work along a Blockstack-like system is implemented on top of an expressive enough blockchain [250] and include Blockstack identities as part of SIAM.

Certificate, key, and software transparency. Bringing transparency to different security-critical domains has been actively studied. Solutions for public-key validation infrastructure are proposed in AKI [145], ARPKI [23] and Certificate Transparency (CT) [163] in which all issued public-key certificates are publicly logged and validated by auditors. EthIKS [40] provides stronger auditability to CONIKS [178], an end-user key verification service based on a

verifiable transparency log, by creating a smart contract on Ethereum that provides pro-active security instead of re-active auditing.

Skipchains can be also seen as an extension of timeline entanglement [175], where nodes can additionally crawl a timeline forward to update their trust and also enabling efficient traversal using authenticated skiplists [93]. Finally, in order to make skipchains more robust to compromised keys, we can combine them with proactive schemes mapped on top of the managing cothority as suggested by Canetti et al. [51].

10 Conclusion and Future Work

We conclude this dissertation by discussing the outcomes of this thesis and its implications, as well as by presenting potential avenues for future research.

10.1 Summary and Implications

This thesis tackles two pressing challenges current blockchain systems repeatedly face. First, the scalability challenge both concerning total capacity or throughput the system can handle and concerning confirmation latency. To understand how pressing this issue is, we consider Bitcoin the most prominent and most valuable Blockchain to-date. Bitcoin has a throughput of 3 transactions per second with a confirmation latency of one hour. Comparing this to VISA that can easily handle 4,000 transactions per second with a confirmation latency of a few seconds we conclude that Bitcoin as a payment system is disappointing, if not unusable, for the average person.

This thesis builds a payment system that can provide a better alternative than Bitcoin to the average person who simply wants to pay his groceries, without the need to trust a central actor such as VISA or a bank. The capstone (scaling wise) of this thesis, OMNILEDGER, has experimentally achieved throughput of 2,500 transactions per second with a confirmation latency of less 10 seconds and has shown potential for far higher throughput, given enough resources. At the same time OMNILEDGER does not assume trust on any centralized actor, making it deployable in environments far more adversarial than what VISA can withstand. A secondary scalability-benefit of OMNILEDGER that might in practice be more important than the first, is that it enables "light clients" who neither mine blocks nor store the entire blockchain history to verify quickly and efficiently that a transaction has committed, without requiring active communication with or having to trust any particular full node.

The second pressing challenge is the privacy of data posted on-chain. A blockchain is (by design) a transparent log visible to all the participants. This, however, is a disadvantage when it comes to using blockchains in an environment where confidentiality is necessary, as most

businesses want to keep their data confidential and only selectively disclose them to vetted collaborators. To tackle this problem, we propose CALYPSO that can be applied as an external service to any programmable blockchain. CALYPSO enables a writer to share confidential data with a reader and guarantees that (a) only the reader is authorized to access the data and (b) if the reader accesses the data, the writer will hold an undeniable proof that the access happened. As a result, with CALYPSO, even fully open blockchains can be used by private companies without forcing them to reveal their confidential data. Furthermore, CALYPSO enables the general verifiable management of private data, providing a secure infrastructure for application typically susceptible to frontrunning attacks such as lotteries and decentralized exchanges.

In our journey to solve these challenges, we required tools that work in a large scale and remain robust under Byzantine adversaries. As a result, in this thesis, we also proposed novel tools for scalable consensus, scalable bias-resistant randomness generation, and decentralized long-term relationship tracking. Although these tools are introduced in the scope of this thesis, their applicability exceeds it. For example, scalable consensus can be a building block for large-scale state-machine replication, bias-resistant randomness can help run public-verifiable lotteries or scale anonymity systems, and decentralized relationship tracking is already proposed as a solution to secure software-update dispersion [195].

10.2 Future Work

We now discuss some potential directions for future research that builds on the contributions of this dissertation and addresses some of its limitations.

10.2.1 Sharding for Smart Contracts

In this dissertation we presented OMNILEDGER, a decentralized global payment network that employs sharding in order to scale out. However, one of the most promising blockchain applications is smart contracts, or self-executed programs, which OMNILEDGER does not support. In future work, we envision a system similar to OMNILEDGER that supports smart contracts. In order to implement this, we want to design a UTXO or account-based smart contract language that supports efficient cross-shard operation without compromising the security guarantees and especially the serializability of transactions that all blockchain systems currently provide.

10.2.2 Heterogeneous Sharding

While showing great promise, smart contracts are difficult to program correctly, as they need a deep understanding of cryptographic and distributed algorithms, and offer limited functionality, as they have to be deterministic and cannot operate on secret data. In future

work, we envision a system called Protean [8] which provides a novel Byzantine fault-tolerant decentralized computer that draws inspiration from the microservice architecture of cloud computing and modern CPU architectures. Protean uses secure, specialized modules, called functional units, for building decentralized applications that are commonly seen in smart contracts, such as lotteries and e-voting schemes. Such modules can be the systems introduced in this dissertation, for example, a Randhound can be a randomness generation functional unit, CALYPSO can be a secret managing functional unit while OMNILEDGER can be a secure and transparent meta-data storage functional unit.

As these functional units are special purpose, they are easier to prove correct than Turing-complete smart contracts and provide a layer of abstraction to smart contract developers who might lack a full understanding of cryptography and distributed algorithms. Furthermore, by hosting arbitrarily-defined functional units, as well as multiple instantiations of the same, abstract functional unit with different security and performance properties, Protean equips smart-contract developers with great flexibility in building their applications, thereby extending the range of applications that are enabled by smart contracts.

10.2.3 Locality Preserving Sharding

While decentralized blockchains are getting increasingly attractive due to their third-party independence, they still suffer from fundamental limitations to increase throughput and lower latency, and to remain secure under attacks. There is abundant work to fix the security and scalability problems of Bitcoin (and of blockchains that inherit some of its design decisions). Existing solutions (including the ones presented in this thesis), however, still do not achieve real-time latencies, especially under heavy load, or they are insecure under strong network adversaries.

One of the main reasons this problem persists is that transactions have to first be announced across the globe and only then get finalized by the validators. OMNILEDGER provided a solution to the problem of forcing every single validator to see the transaction and instead requires only a subset (or shard) to consent. However, because of the random selection of validators per shard there is still a high chance that the transaction has to travel across the globe and back.

To mitigate this issue, we envision Blinkchain [21], a latency-aware blockchain that provides bounds on consensus latency. We retain sharding as a technique to scale horizontally. Our proposal differs in the way we select validators for the blockchain in each shard. Namely, to achieve low-latency consensus among validators, we use techniques from Crux [22], which enhance scalable distributed protocols with low latency.

10.2.4 Alternatives to Proof-of-Work

Bitcoin and many of its offspring use proof-of-work (PoW) mechanisms [16] to allow pseudonymous, untrusted, external actors to securely extend the blockchain. However, PoW requires costly special-purpose hardware and consumes massive amounts of electricity. This has led to a re-centralization since only a few privileged entities who have access to the necessary resources are capable to mine, whereas regular users who can not afford such hardware and its maintenance are excluded. Consequently, the control over the entire system rests in the hands of a small number of elite users, for example as in Bitcoin; an undemocratic approach.

To mitigate this problem a first step is to study Proof-of-stake (PoS), where participants use their assets (coins) to create (mint) new assets, and which promises similar properties as PoW but consumes far less energy. As future work, we intend to combine the building blocks that are introduced in this thesis to create a functional PoS system.

However, PoS is essentially nothing but a shareholder corporation where the rich again have an advantage as they possess more assets and thus are able to mint new coins faster than less-privileged participants. As a consequence, the (already) rich become even richer; again, an undemocratic approach. This is why we developed *proof-of-personhood (PoP)* [43] as a first step towards our goal, which combines *pseudonym parties* [105] with state-of-the-art cryptographic tools like linkable ring signatures [171] and collective signing [243] to create so-called *PoP-tokens*, which are basically *accountable anonymous credentials*.

The core idea of pseudonym parties is to verify *real people*, thereby linking physical and virtual identities and providing a basis to prevent adversaries from mounting Sybil attacks. Pseudonym parties are, as the name suggests, parties which can be organized basically by anyone, from governments to non-profit organizations, or companies to small groups of people in their own village. The participants agree on a set of rules such as specifying a place and time. All parties are recorded for transparency, but attendees are free to hide their identities by dressing as they wish, including hiding their faces for anonymity. By the end of the party, each attendee will obtain *exactly one* cryptographic identity token that represents both a physical and virtual identity without revealing sensitive information.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine Fault-tolerant Services. *SIGOPS Operating Systems Review*, 39(5):59–74, October 2005.

[2] Joe Abley, David Blacka, David Conrad, Richard Lamb, Matt Larson, Fredrik Ljunggren, David Knight, Tomofumi Okubo, and Jakob Schlyter. DNSSEC Root Zone – High Level Technical Architecture, June 2010. Version 1.4.

[3] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In James Aspnes, Alysson Bessani, Pascal Felber, and João Leitão, editors, *21st International Conference on Principles of Distributed Systems (OPODIS 2017)*, volume 95 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:19, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] Ben Adida. Helios: Web-based Open-audit Voting. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

[5] Akamai Technologies, Inc. Akamai: Content Delivery Network (CDN) & Cloud Computing. May 2016.

[6] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. Chainspace: A sharded smart contracts platform. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[7] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J. Freedman. Blockstack: A Global Naming and Storage System Secured by Blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 181–194, Denver, CO, June 2016. USENIX Association.

[8] Enis Ceyhun Alp, Eleftherios Kokoris-Kogias, Georgia Fragkouli, and Bryan Ford. Rethinking General-Purpose Decentralized Computing. In *XVII Workshop on Hot Topics in Operating Systems (HotOS)*, May 2019.

[9] Gavin Andresen. Classic? Unlimited? XT? Core?, January 2016.

[10] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the Thirteenth European conference on Computer systems*, EuroSys '18, New York, NY, USA, 2018. ACM.

[11] Elli Androulaki, Christian Cachin, Angelo De Caro, and Eleftherios Kokoris-Kogias. Channels: Horizontal scaling and confidentiality on permissioned blockchains. In *European Symposium on Research in Computer Security*, pages 111–131. Springer, 2018.

[12] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Secure multiparty computations on bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 443–458, 2014.

[13] Maria Apostolaki, Aviv Zohar, and Laurent Vanbever. Hijacking Bitcoin: Large-scale Network Attacks on Cryptocurrencies. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 375–392, 2017.

[14] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of Space: When Space is of the Essence. In *Security and Cryptography for Networks*, pages 538–557. Springer, 2014.

[15] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. MedRec: Using blockchain for medical data access and permission management. In *Open and Big Data (OBD), International Conference on*, pages 25–30. IEEE, 2016.

[16] Adam Back. Hashcash – A Denial of Service Counter-Measure, August 2002.

[17] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. Enabling Blockchain Innovations with Pegged Sidechains. October 2014.

[18] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. Trap Me If You Can – Million Dollar Curve. Cryptology ePrint Archive, Report 2015/1249, 2015.

[19] James Ball and Dominic Rushe. NSA Prism program taps in to user data of Apple, Google and others. October 2013.

[20] Devlin Barret. U.S. Suspects Hackers in China Breached About 4 Million People's Records, Officials Say. June 2015.

[21] Cristina Basescu, Eleftherios Kokoris-Kogias, and Bryan Alexander Ford. Low-latency Blockchain Consensus. Technical report, May 2017.

[22] Cristina Basescu, Michael F Nowlan, Kirill Nikitin, Jose M Faleiro, and Bryan Ford. Crux: Locality-Preserving Distributed Services. *arXiv preprint arXiv:1405.0637*, 2014.

[23] David A. Basin, Cas J. F. Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. ARPKI: attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 382–393, 2014.

[24] Kevin S. Bauer, Damon McCoy, Dirk Grunwald, and Douglas C. Sicker. BitBlender: Light-Weight Anonymity for BitTorrent. In *4th International ICST Conference on Security and Privacy in Communication Networks, SECURECOMM 2008, Istanbul, Turkey, September 22-25, 2008*, 2008.

[25] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006*, pages 207–228, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[26] Daniel J. Bernstein, Tung Chou, Chitchanok Chuengsatiansup, Andreas Hülsing, Eran Lambooij, Tanja Lange, Ruben Niederhagen, and Christine van Vredendaal. How to manipulate curve standards: A white paper for the black hat. In *Security Standardisation Research - Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, pages 109–139, 2015.

[27] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012.

[28] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980, 2013.

[29] Daniel J. Bernstein, Tanja Lange, and Ruben Niederhagen. Dual EC: A standardized back door. In *The New Codebreakers - Essays Dedicated to David Kahn on the Occasion of His 85th Birthday*, pages 256–281, 2016.

[30] Bitcoin Wiki. Confirmation, en.bitcoin.it/wiki/confirmation, accessed may 2016.

[31] Bitcoin Wiki. Scalability, en.bitcoin.it/wiki/scalability, accessed may 2016, 2016.

[32] Bitnodes. Bitcoin Network Snapshot, accessed april 2017, April 2017.

[33] George Robert Blakley et al. Safeguarding cryptographic keys. In *Proceedings of the national computer conference*, volume 48, pages 313–317, 1979.

[34] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 127–144. Springer, 1998.

[35] Blockchain.info. Blockchain Size, blockchain.info/charts/blocks-size, February 2017.

[36] Manuel Blum. Coin flipping by telephone. In *Advances in Cryptology: A Report on CRYPTO 81, IEEE Workshop on Communications Security, Santa Barbara, California, USA, August 24-26, 1981.*, pages 11–15, 1981.

[37] Carlo Blundo, Alfredo De Santis, and Ugo Vaccaro. Randomness in distribution protocols. In Serge Abiteboul and Eli Shamir, editors, *Automata, Languages and Programming*, volume 820 of *Lecture Notes in Computer Science*, pages 568–579. Springer Berlin Heidelberg, 1994.

[38] Alexandra Boldyreva. Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme. In *Public Key Cryptography – PKC 2003*. Springer, 2002.

[39] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 514–532. Springer, 2001.

[40] Joseph Bonneau. EthIKS: Using Ethereum to Audit a CONIKS Key Transparency Log. In *Financial Cryptography and Data Security 2016*. Springer Berlin Heidelberg, 2016.

[41] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On Bitcoin as a public randomness source. IACR eprint archive, October 2015.

[42] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua Kroll, and Edward W Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. In *2015 IEEE Symposium on Security and Privacy. IEEE*, 2015.

[43] Maria Borge, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, and Bryan Ford. Proof-of-Personhood: Redemocratizing Permissionless Cryptocurrencies. In *1st IEEE Security and Privacy on the Blockchain*, April 2017.

[44] Peter Bright. How the Comodo certificate fraud calls CA trust into questions. *arstechnica*, March 2011.

[45] Peter Bright. Another fraudulent certificate raises the same old questions about certificate authorities. *arstechnica*, August 2011.

[46] Ethan Buchman. Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, 2016.

[47] Vitalik Buterin, Jeff Coleman, and Matthew Wampler-Doty. Notes on Scalable Blockchain Protocols (version 0.3), 2015.

[48] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, pages 524–541, 2001.

[49] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random Oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA.*, pages 123–132, 2000.

[50] Christian Cachin and Marko Vukolic. Blockchain Consensus Protocols in the Wild (Keynote Talk). In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[51] Ran Canetti, Shai Halevi, and Amir Herzberg. Maintaining authenticated communication in the presence of break-ins. *Journal of Cryptology*, 13(1):61–105, Jan 2000.

[52] Justin Cappos, Justin Samuel, Scott M. Baker, and John H. Hartman. A Look In the Mirror: Attacks on Package Managers. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 565–574, 2008.

[53] Ignacio Cascudo and Bernardo David. SCRAPE: scalable randomness attested by public entities. In *Applied Cryptography and Network Security - 15th International Conference, ACNS 2017, Kanazawa, Japan, July 10-12, 2017, Proceedings*, pages 537–556, 2017.

[54] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. SplitStream: high-bandwidth multicast in cooperative environments. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[55] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

[56] David Chaum and Torben P. Pedersen. Wallet databases with observers. In *IACR International Cryptology Conference (CRYPTO)*, 1992.

[57] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[58] Richard Chirgwin. iOS 7's weak random number generator stuns kernel security. *The Register*, March 2014.

[59] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *Symposium on Foundations of Computer Science (SFCS)*, SFCS '85, pages 383–395, Washington, DC, USA, 1985. IEEE Computer Society.

[60] Jeremy Clark and Urs Hengartner. On the Use of Financial Data as a Random Beacon. In *2010 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections, EVT/WOTE '10, Washington, D.C., USA, August 9-10, 2010*, 2010.

[61] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, July 2000.

[62] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *6th USENIX Symposium on Networked Systems Design and Implementation*, April 2009.

[63] CloudFlare, Inc. CloudFlare - The web performance & security company. May 2016.

[64] CoinDesk. Decentralized Exchanges Aren't Living Up to Their Name - And Data Proves It, July 2018.

[65] Comodo Group. Comodo Fraud Incident. July 2015.

[66] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, May 2008.

[67] 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 Computer Systems*, 31(3):8:1–8:22, August 2013.

[68] Henry Corrigan-Gibbs, Wendy Mu, Dan Boneh, and Bryan Ford. Ensuring high-quality randomness in cryptographic key generation. In *20th ACM Conference on Computer and Communications Security (CCS)*, November 2013.

[69] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[70] Kyle Croman, Christian Decke, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gun Sirer, Dawn Song an, and Roger Wattenhofer. On Scaling Decentralized Blockchains (A Position Paper). In *3rd Workshop on Bitcoin and Blockchain Research*, 2016.

[71] Scott A. Crosby and Dan S. Wallach. Efficient data structures for tamper-evident logging. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 317–334, 2009.

[72] CuriosMind. World's Hottest Decentralized Lottery Powered by Blockchain, February 2018.

[73] Matt Czernik. On Blockchain Frontrunning , February 2018.

[74] George Danezis and Sarah Meiklejohn. Centrally Banked Cryptocurrencies. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[75] Debian. Advanced Package Tool, May 2016.

[76] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[77] Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin Meets Strong Consistency. In *17th International Conference on Distributed Computing and Networking (ICDCN), Singapore*, January 2016.

[78] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Stabilization, Safety, and Security of Distributed Systems*, pages 3–18. Springer, August 2015.

[79] Sebastiaan Deetman. Bitcoin Could Consume as Much Electricity as Denmark by 2020, May 2016.

[80] DeterLab Network Security Testbed, September 2012.

[81] Tim Dierks and Eric Rescorla. RFC 5246-the transport layer security (TLS) protocol version 1.2. *Internet Engineering Task Force*, 2008.

[82] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: the second-generation onion router. In *12th USENIX Security Symposium*, August 2004.

[83] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 202–210. ACM, 2003.

[84] John R. Douceur. The Sybil Attack. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[85] Manu Drijvers, Kasra Edalatnejad, Bryan Ford, Eike Kiltz, Julian Loss, Gregory Neven, and Igors Stepanovs. On the security of two-round multi-signatures. In *IEEE Symposium on Security and Privacy*. IEEE, May 2019.

[86] Alevtina Dubovitskaya, Zhigang Xu, Samuel Ryu, Michael Schumacher, and Fusheng Wang. Secure and Trustable Electronic Medical Records Sharing using Blockchain. 2017.

[87] D. Eastlake. Domain name system security extensions, March 1999. RFC 2535.

[88] Rachid El Bansarkhani and Mohammed Meziani. An efficient lattice-based secret sharing construction. In *IFIP International Workshop on Information Security Theory and Practice*, pages 160–168. Springer, 2012.

[89] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1985.

[90] Michael Elkins, David Del Torto, Raph Levien, and Thomas Roessler. MIME security with OpenPGP. Technical report, 2001. RFC 3156.

[91] Justin Ellis. The Guardian introduces SecureDrop for document leaks. *Nieman Journalism Lab*, 2014.

[92] Elmootazbellah Nabil Elnozahy, David B. Johnson, and Willy Zwaenepoel. The Performance of Consistent Checkpointing. In *11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE, 1992.

[93] C. Christopher Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. In *Proceedings of the 2009 ACM Conference on Computer and Communications Security, CCS 2009, Chicago, Illinois, USA, November 9-13, 2009*, pages 213–222, 2009.

[94] European Parliament and Council of the European Union. General Data Protection Regulation (GDPR). *Official Journal of the European Union (OJ)*, L119:1–88, 2016.

[95] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, Santa Clara, CA, March 2016. USENIX Association.

[96] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[97] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[98] Joan Feigenbaum. Multiple Objectives of Lawful-Surveillance Protocols (Transcript of Discussion). In *Cambridge International Workshop on Security Protocols*, pages 9–17. Springer, 2017.

[99] Paul Feldman. A practical scheme for non-interactive verifiable secret sharing. In *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987*, pages 427–437, 1987.

[100] Amos Fiat and Adi Shamir. How to prove yourself: practical solutions to identification and signature problems. In *IACR International Cryptology Conference (CRYPTO)*, pages 186–194, 1987.

[101] Hal Finney. Best practice for fast transaction acceptance – how high is the risk?, February 2011. Bitcoin Forum comment.

[102] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[103] Bryan Ford. Apple, FBI, and Software Transparency. *Freedom to Tinker*, March 2016.

[104] Bryan Ford et al. Persistent personal names for globally connected mobile devices. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[105] Bryan Ford and Jacob Strauss. An offline foundation for online accountable pseudonyms. In *1st International Workshop on Social Network Systems (SocialNets)*, 2008.

[106] Matthew Franklin and Haibin Zhang. Unique ring signatures: A practical construction. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security 2013*, pages 162–170. Springer Berlin Heidelberg, 2013.

[107] Matthew K Franklin and Haibin Zhang. A Framework for Unique Ring Signatures. *IACR Cryptology ePrint Archive*, 2012:577, 2012.

[108] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: Increasing Data Privacy with Self-Destructing Data. In *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, pages 299–316, 2009.

[109] Genecoin. Make a Backup of Yourself Using Bitcoin, May 2018.

[110] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. In *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, pages 295–310, 1999.

[111] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.

[112] Arthur Gervais, Ghassan O. Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the Security and Performance of Proof of Work Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 3–16, 2016.

[113] Arthur Gervais, Hubert Ritzdorf, Ghassan O Karame, and Srdjan Capkun. Tampering with the Delivery of Blocks and Transactions in Bitcoin. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 692–705. ACM, 2015.

[114] Mainak Ghosh, Miles Richardson, Bryan Ford, and Rob Jansen. A TorPath to TorCoin: Proof-of-bandwidth altcoins for compensating relays. In *Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs)*, 2014.

[115] Samuel Gibbs. Man hacked random-number generator to rig lotteries, investigators say. *The Guardian*, April 2016.

[116] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68, 2017.

[117] Sharad Goel, Mark Robson, Milo Polte, and Emin Gun Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. Technical Report 2003-1890, Cornell University, February 2003.

[118] The Go Programming Language, February 2018.

[119] David Goulet and George Kadianakis. Random Number Generation During Tor Voting, 2015.

[120] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[121] Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. Highly Dynamic Distributed Computing with Byzantine Failures. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 176–183, New York, NY, USA, 2013. ACM.

[122] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *5th European conference on Computer systems*, pages 363–376. ACM, 2010.

[123] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. Analysis of the Linux Random Number Generator. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pages 371–385, 2006.

[124] J. Alex Halderman and Brent Waters. Harvesting Verifiable Challenges from Oblivious Online Sources. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 330–341, New York, NY, USA, 2007. ACM.

[125] Timo Hanke and Dominic Williams. Intoducing Random Beascons Using Threshold Relay Chains, September 2016.

[126] Ed Hardt. The OAuth 2.0 authorization framework, October 2012. RFC 6749.

[127] Mike Hearn and J Spilman. Rapidly-adjusted (micro)payments to a pre-determined party, 2015.

[128] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse Attacks on Bitcoin's Peer-to-Peer Network. In *24th USENIX Security Symposium*, pages 129–144, 2015.

[129] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. Proactive secret sharing or: How to cope with perpetual leakage. *Advances in Cryptology—CRYPT0'95*, pages 339–352, 1995.

[130] Kate Fultz Hollis. To Share or Not to Share: Ethical Acquisition and Use of Medical Data. *AMIA Summits on Translational Science Proceedings*, 2016:420, 2016.

[131] M. Horton and R. Adams. Standard for interchange of USENET messages, December 1987. RFC 1036.

[132] Max Howell. Homebrew – The Missing Packet Manager for macOS, May 2016.

[133] Longxia Huang, Gongxuan Zhang, Shui Yu, Anmin Fu, and John Yearwood. SeShare: Secure cloud data sharing based on blockchain and public auditing. *Concurrency and Computation: Practice and Experience*.

[134] Emin Gün Sirer Ittay Eyal. It's Time For a Hard Bitcoin Fork, June 2014.

[135] Markus Jakobsson. On quorum controlled asymmetric proxy re-encryption. In *Public key cryptography*, pages 632–632. Springer, 1999.

[136] Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. On technical security issues in cloud computing. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 109–116. IEEE, 2009.

[137] Audun Jøsang, Roslan Ismail, and Colin Boyd. A survey of trust and reputation systems for online service provision. *Decision support systems*, 43(2):618–644, 2007.

[138] HIPAA Journal. The Benefits of Using Blockchain for Medical Records, September 2017.

[139] IBM Blockchain Juan Delacruz. Blockchain is tackling the challenge of data sharing in government, May 2018.

[140] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in Bitcoin. In *19th ACM Conference on Computer and communications security*, pages 906–917. ACM, 2012.

[141] Aniket Kate and Ian Goldberg. Distributed Key Generation for the Internet. In *Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on*, pages 119–128. IEEE, 2009.

[142] Idit Keidar and Danny Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 245–254. ACM, 1995.

[143] Limor Kessem. Certificates-as-a-Service? Code Signing Certs Become Popular Cybercrime Commodity, September 2015.

[144] Katherine K Kim, Pamela Sankar, Machelle D Wilson, and Sarah C Haynes. Factors affecting willingness to share electronic health data among California consumers. *BMC medical ethics*, 18(1):25, 2017.

[145] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable Key Infrastructure (AKI): A Proposal for a Public-Key Validation Infrastructure. In *International Word Wide Web Conference (WWW)*, 2014.

[146] Yongdae Kim, Adrian Perrig, and Gene Tsudik. Tree-based group key agreement. *ACM Transactions on Information and System Security (TISSEC)*, 7(1):60–96, 2004.

[147] Sunny King and Scott Nadal. PPCoin: Peer-to-peer Crypto-Currency with Proof-of-Stake. August 2012.

[148] Eleftherios Kokoris-Kogias, Linus Gasser, Ismail Khoffi, Philipp Jovanovic, Nicolas Gailly, and Bryan Ford. Managing Identities Using Blockchains and CoSi. Technical report, 9th Workshop on Hot Topics in Privacy Enhancing Technologies (HotPETs 2016), 2016.

[149] 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 *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[150] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 583–598, 2018.

[151] Eleftherios Kokoris-Kogias, Orfefs Voutyras, and Theodora Varvarigou. TRM-SIoT: A scalable hybrid trust & reputation model for the social Internet of Things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–9. IEEE, 2016.

[152] Charles R Korsmo. High-Frequency Trading: A Regulatory Strategy. *U. Rich. L. Rev.*, 48:523, 2013.

[153] Ahmed E. Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 839–858, 2016.

[154] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM, October 2007.

[155] Maxwell N Krohn, Michael J Freedman, and David Mazières. On-the-fly verification of rateless erasure codes for efficient content distribution. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 226–240. IEEE, 2004.

[156] Bogdan Kulynych, Wouter Lueks, Marios Isaakidis, George Danezis, and Carmela Troncoso. ClaimChain: Improving the Security and Privacy of In-band Key Distribution for Messaging. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 86–103, 2018.

[157] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. How to use Bitcoin to play decentralized poker. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 195–206. ACM, 2015.

[158] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using Delegations to Protect Community Repositories. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2016.

[159] Jae Kwon. TenderMint: Consensus without Mining. 2014.

[160] The Kyber Cryptography Library, 2010 – 2018.

[161] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[162] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[163] Ben Laurie. Certificate Transparency. *ACM Queue*, 12(8), September 2014.

[164] Timothy B. Lee. Facebook's Cambridge Analytica Scandal, Explained [Updated], 2018 (accessed July 27, 2018).

[165] Arjen K. Lenstra and Benjamin Wesolowski. Trustworthy public randomness with sloth, unicorn, and trx. *IJACT*, 3(4):330–343, 2017.

[166] Chris Lesniewski-Laas and M. Frans Kaashoek. Whanau: A Sybil-proof distributed hash table. In *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*, pages 111–126, 2010.

[167] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[168] Jinyuan Li, Maxwell N. Krohn, David Mazières, and Dennis E. Shasha. Secure untrusted data repository (SUNDR). In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*, pages 121–136, 2004.

[169] Jian Liang, Naoum Naoumov, and Keith W. Ross. The index poisoning attack in P2P file sharing systems. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 23-29 April 2006, Barcelona, Catalunya, Spain*, 2006.

[170] Laure A Linn and Martha B Koo. Blockchain for health data and its potential use in health it and health care related research. In *ONC/NIST Use of Blockchain for Healthcare and Research Workshop. Gaithersburg, Maryland, United States: ONC/NIST*, 2016.

[171] Joseph K. Liu, Victor K. Wei, and Duncan S. Wong. Linkable spontaneous anonymous group signature for ad hoc groups. In *Australian Conference on Information Security and Privacy*, pages 614–623, July 2004.

[172] T. Lodderstedt, S. Dronia, and M. Scurtescu. OAuth 2.0 token revocation, August 2013. RFC 7009.

[173] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A Secure Sharding Protocol For Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.

[174] Aanchal Malhotra, Isaac E. Cohen, Erik Brakke, and Sharon Goldberg. Attacking the Network Time Protocol. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[175] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 297–312, 2002.

[176] David Mazières. The Stellar Consensus Protocol: A Federated Model for Internet-level Consensus. February 2016.

[177] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 108–117. ACM, 2002.

[178] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing Key Transparency to End Users. In *Proceedings of the 24th USENIX Conference on Security Symposium*, pages 383–398. USENIX Association, 2015.

[179] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, June 1979.

[180] Silvio Micali, Salil Vadhan, and Michael Rabin. Verifiable random functions. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 120–130. IEEE Computer Society, 1999.

[181] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 11–11. USENIX Association, 2007.

[182] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-Cash from Bitcoin. In *34th IEEE Symposium on Security and Privacy (S&P)*, May 2013.

[183] Andrew Miller and Iddo Bentov. Zero-collateral lotteries in Bitcoin and Ethereum. In *Security and Privacy Workshops (EuroS&PW), 2017 IEEE European Symposium on*, pages 4–13. IEEE, 2017.

[184] Andrew Miller and Rob Jansen. Shadow-Bitcoin: scalable simulation via direct execution of multi-threaded applications. In *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, 2015.

[185] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, New York, NY, USA, 2016. ACM.

[186] Michael Mimoso. D-Link Accidentally Leaks Private Code-Signing Keys. *ThreatPost*, September 2015.

[187] Mininet – An Instant Virtual Network on your Laptop (or other PC), February 2018.

[188] Paul V. Mockapetris and Kevin J. Dunlap. Development of the domain name system. In *SIGCOMM '88, Proceedings of the ACM Symposium on Communications Architectures and Protocols, Stanford, CA, USA, August 16-18, 1988*, pages 123–133, 1988.

[189] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. Elastic management of cluster-based services in the cloud. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, pages 19–24, New York, NY, USA, 2009. ACM.

[190] J. Ian Munro, Thomas Papadakis, and Robert Sedgewick. Deterministic Skip Lists. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '92, pages 367–375, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[191] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[192] National Institute of Standards and Technology. *Security of Interactive and automated Access Management Using Secure Shell (SSH)*, chapter 5.1.2. 2015.

[193] National Institute of Standards and Technology. NIST Randomness Beacon, 2017.

[194] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn Mining: Generalizing Selfish Mining and Combining with an Eclipse Attack. In *1st IEEE European Symposium on Security and Privacy*, March 2015.

[195] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287. USENIX Association, 2017.

[196] Olumuyiwa Oluwasanmi and Jared Saia. Scalable Byzantine Agreement with a Random Beacon. In Andréa W. Richa and Christian Scheideler, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 7596 of *Lecture Notes in Computer Science*, pages 253–265. Springer Berlin Heidelberg, 2012.

[197] Henning Pagnia and Felix C Gärtner. On the impossibility of fair exchange without a trusted third party. Technical report, Technical Report TUD-BS-1999-02, Darmstadt University of Technology, Department of Computer Science, Darmstadt, Germany, 1999.

[198] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[199] Rafael Pass, Cornell Tech, and Lior Seeman. Analysis of the Blockchain Protocol in Asynchronous Networks. *Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, 2017.

[200] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, 1980.

[201] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, pages 129–140, 1991.

[202] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings*, pages 522–526, 1991.

[203] Marc Pilkington. Blockchain technology: principles and applications. *Browser Download This Paper*, 2015.

[204] Andrew Poelstra, Adam Back, Mark Friedenbach, Gregory Maxwell, and Pieter Wuille. Confidential assets. In *International Conference on Financial Cryptography and Data Security*, pages 43–63. Springer, 2018.

[205] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments, January 2016.

[206] Serguei Popov. On a decentralized trustless pseudo-random number generation algorithm. *J. Mathematical Cryptology*, 11(1):37–43, 2017.

[207] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*, 33(6):668–676, June 1990.

[208] Python Community. PyPI - the Python Package Index, September 2016.

[209] Python Community. EasyInstall Module, May 2016.

[210] Michael O. Rabin. Transaction Protection by Beacons. *Journal of Computer and System Sciences*, 27(2):256–267, 1983.

[211] Jean Louis Raisaro, Juan Troncoso-Pastoriza, Mickaël Misbach, João Sá Sousa, Sylvain Pradervand, Edoardo Missiaglia, Olivier Michielin, Bryan Ford, and Jean-Pierre Hubaux. MedCo: Enabling secure and privacy-preserving exploration of distributed clinical and genomic data. *IEEE/ACM transactions on computational biology and bioinformatics*, 2018.

[212] randao.org. Randao: Verifiable Random Number Generation, 2017.

[213] Michael K Reiter and Kenneth P Birman. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):986–1009, 1994.

[214] R.L. Rivest and B. Lampson. SDSI: A Simple Distributed Security Infrastructure, April 1996. http://theory.lcs.mit.edu/~cis/sdsi.html.

[215] Ronald L Rivest, Adi Shamir, and David A Wagner. Time-lock puzzles and timed-release crypto. 1996.

[216] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[217] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable Key Compromise in Software Update Systems. In *17th ACM Conference on Computer and Communications security (CCS)*, October 2010.

[218] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from Bitcoin. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 459–474. IEEE, 2014.

[219] Satoshi.info. Unspent Transaction Output Set, February 2017.

[220] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[221] Claus P. Schnorr. Efficient signature generation by smart cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[222] Berry Schoenmakers. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *IACR International Cryptology Conference (CRYPTO)*, pages 784–784, 1999.

[223] Adam Schwartz and Cindy Cohn. "Information Fiduciaries" Must Protect Your Data Privacy, October 2018.

[224] David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, page 5, 2014.

[225] SECBIT. How the winner got Fomo3D prize — A Detailed Explanation, August 2018.

[226] Hossein Shafagh, Lukas Burkhalter, Anwar Hithnawi, and Simon Duquennoy. Towards Blockchain-based Auditable Storage and Sharing of IoT Data. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, pages 45–50. ACM, 2017.

[227] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, 1979.

[228] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[229] Victor Shoup and Rosario Gennaro. Securing threshold cryptosystems against chosen ciphertext attack. *Advances in Cryptology—EUROCRYPT'98*, pages 1–16, 1998.

[230] Dan Shumow and Niels Ferguson. On the Possibility of a Back Door in the NIST SP800-90 Dual EC PRNG. CRYPTO 2007 Rump Session, 2007. http://rump2007.cr.yp.to/15-shumow.pdf.

[231] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *1st International Workshop on Peer-to-Peer Systems (IPTPS)*, March 2002.

[232] Matthew Skala. Hypergeometric Tail Inequalities: Ending the Insanity. *CoRR*, abs/1311.5939, 2013.

[233] Yonatan Sompolinsky and Aviv Zohar. Accelerating Bitcoin's Transaction Processing. Fast Money Grows on Trees, Not Chains, December 2013.

[234] Markus Stadler. Publicly Verifiable Secret Sharing. In *15th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 190–199, Berlin, Heidelberg, 1996. Springer.

[235] Douglas R. Stinson and Reto Strobl. Provably secure distributed Schnorr signatures and a (t, n) threshold scheme for implicit certificates. In Vijay Varadharajan and Yi Mu, editors, *Australasian Conference on Information Security and Privacy (ACISP)*, pages 417–434, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[236] Ion Stoica, Robert Tappan Morris, David R. Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, pages 149–160, 2001.

[237] Hemang Subramanian. Decentralized blockchain-based electronic marketplaces. *Communications of the ACM*, 61(1):78–84, 2017.

[238] Apple Support. Frequently asked questions about iCloud Keychain, May 2016.

[239] Melanie Swan. *Blockchain: Blueprint for a new economy*. O'Reilly Media, Inc., 2015.

[240] Tim Swanson. Consensus-as-a-service: a brief report on the emergence of permissioned, distributed ledger systems. April 2015.

[241] Martin Holst Swende. Blockchain Frontrunning , October 2017.

[242] 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 *38th IEEE Symposium on Security and Privacy*, May 2017.

[243] 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 *37th IEEE Symposium on Security and Privacy*, May 2016.

[244] Simon Thomsen. Ashley Madison data breach. July 2015.

[245] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 137–152, New York, NY, USA, 2015. ACM.

[246] Marko Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *International Workshop on Open Problems in Network Security*, pages 112–125. Springer, 2015.

[247] Bitcoin Wiki. Proof of burn, September 2017.

[248] Wikipedia. Atomic commit, February 2017.

[249] David Isaac Wolinsky, Henry Corrigan-Gibbs, Aaron Johnson, and Bryan Ford. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[250] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. *Ethereum Project Yellow Paper*, 2014.

[251] David Yermack. Corporate governance and blockchains. *Review of Finance*, 21(1):7–31, 2017.

[252] T. Ylonen and C. Lonvick, Ed. The secure shell protocol architecture, January 2006. RFC 4251.

[253] Ernst & Young. Blockchain in health, May 2018.

[254] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. SybilLimit: A Near-Optimal Social Network Defense against Sybil Attacks. In *29th IEEE Symposium on Security and Privacy (S&P)*, May 2008.

[255] YUM. Yellowdog Updater Modified, May 2016.

[256] Guy Zyskind, Oz Nathan, et al. Decentralizing privacy: Using blockchain to protect personal data. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 180–184. IEEE, 2015.

[257] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471*, 2015.

Index

Lefteris Kokoris-Kogias

Researcher (PhD) on decentralized systems and applied cryptography

Chemin du Croset 1c, Lausanne, Switzerland
Phone: 0041 78 630 45 57
E-mail:eleftherios.kokoriskogias@epfl.ch
Links: [scholar], [epfl], [LinkedIn]

RESEARCH INTERESTS

Decentralized systems and algorithms (focus on blockchain technology); Privacy-preserving technologies (focus on anonymous communications and confidential data-sharing); Electronic voting; Secure, reliable, and authenticated information dissemination.

EXPERIENCE

2015-2019 **Research Scientist (Decentralized and Distributed Systems Lab – EPFL).**

- Scalable Byzantine fault tolerance with focus on Blockchain technologies.
- Scalable, bias-resistant randomness creation.
- Distributed, secure, privacy hardened data management.
- Secure decentralized software updates.
- Teaching assistant and Bachelor/ Master/ Junior PhD projects supervision.

01-04/2018 **Member of Research Team**
(Blockchain and Industrial Platforms Lab – IBM Research Zurich).

- Research on novel capabilities for Hyperledger Fabric.
- Resulted in publication and partial integration in Hyperledger Fabric.

2014-2015 **Research Scientist (Distributed Knowledge and Media Systems Group – NTUA).**

- Designed a scalable Trust & Reputation algorithm for the Internet of Things.
- Resulted in publication and included in COSMOS (EU FP7) final deliverables.

EDUCATION

2015-2019 **PhD in Computer Science.**
EPFL.
Advisor: Prof. Bryan Ford.

2010-2015 **5-Year Diploma in Electrical and Computer Engineering.**
School of Electrical and Computer Engineering National Technical University of Athens.
Grade: **9.18/10** – excellent (top 3%).
Major: Computer Systems & Software & Networks – Grade: **9.8/10** (2nd out of 105).
Minor: Bioengineering.

PUBLICATIONS

1. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing, **Eleftherios Kokoris-Kogias**, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, Bryan Ford, *In 25th USENIX Security Symposium, 2016* (accepted 72/467).
2. Scalable Bias-Resistant Distributed Randomness, Ewa Syta, Philipp Jovanovic, **Eleftherios Kokoris-Kogias,** Nicolas Gailly, Ismail Khoffi, Linus Gasser, Michael J. Fischer, Bryan Ford, *38th IEEE Symposium on Security and Privacy, 2017* (accepted 60/463).
3. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds, Kirill Nikitin, **Eleftherios Kokoris-Kogias,** Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, Justin Cappos, Bryan Ford, *In 26th USENIX Security Symposium, 2017* (accepted 85/522).
4. OmniLedger: A Secure, Scale-Out, Decentralized Ledger, **Eleftherios Kokoris-Kogias**, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Bryan Ford, *39th IEEE Symposium on Security and Privacy, 2018* (accepted 63/549).
5. Channels: Horizontal Scaling and Confidentiality on Permissioned Blockchains: Elli Androulaki, Cristian Cachin, Angelo De Caro, **Eleftherios Kokoris-Kogias***. *European Symposium on Research in Computer Security. Springer, September, 2018.*
6. Rethinking General-Purpose Decentralized Computing, Enis Ceyhun Alp*, **Eleftherios Kokoris-Kogias***, Georgia Fragkouli, Bryan Ford, *17th Workshop on Hot Topics in Operating Systems (HotOS), 2019* (accepted 30/125)
7. Proof-of-Personhood: Redemocratizing permissionless cryptocurrencies*,* Maria Borge, **Eleftherios Kokoris-Kogias**, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Bryan Ford, *1st IEEE Security and Privacy On The Blockchain, 2017.*
8. Managing Identities Using Blockchains and CoSi, **Eleftherios Kokoris-Kogias,** Linus Gasser, Ismail Khoffi, Philipp Jovanovic, Nicolas Gailly, Bryan Ford, *9th Hot Topics in Privacy Enhancing Technologies (HotPETs), 2016* (accepted 9/29)
9. TRM-SIoT: A Scalable Hybrid Trust & Reputation Model for the Social Internet of Things, **Eleftherios Kokoris-Kogias,** Orfefs Voutyras,Theodora Varvarigou, *21st International Conference on Emerging Technologies and Factory Automation (ETFA), September 2016*
10. Cryptographically Verifiable Data Structure Having Multi-Hop Forward and Backwards Links and Associated Systems and Methods. **Eleftherios Kokoris-Kogias**, Philipp Jovanovic, Linus Gasser and Bryan Ford. *US-Patent Application. May 2017*
11. Hidden in Plain Sight: Storing and Managing Secrets on a Public Ledger. **Eleftherios Kokoris-Kogias**, Enis Ceyhun Alp, Sandra Deepthy Siby, Nicolas Gailly, Philipp Jovanovic, Linus Gasser, Bryan Ford. *Under Submission. February 2018*
12. Methods and Systems for Secure Data Exchange: **Eleftherios Kokoris-Kogias**, Philipp Jovanovic, Linus Gasser, Bryan Ford. *PCT-Patent Application, February 2018*

AWARDS

- IBM PhD Fellowship 2017-2018, 2018-2020.
- EPFL Appreciation for Exceptional Performance 2017, 2018
- Winner of SICPA workshop “Granting a product proof of origin in a complex value chain” 2017.
- EPFL IC School Fellowship for the Doctoral Program 2015-2016.
- Thomaidion Award for Academic Excellence NTUA 2014-2015.
- Kary Award nomination for excellent academic performance, 2013-2014, 2014-2015.
- 2nd in Greece, IEEE Xtreme 7.0 24h Programming Competition 2013.

INVITED TALKS

Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing.
USENIX Security Conference – August 2016, Scaling Bitcoin Workshop – October 2016
Geneva Bitcoin Meetup – January 2017, Master Workshop 2.0. – November 2018

Scalable Bias-Resistant Distributed Randomness.
IBM Research Zurich – November 2017

OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding.
IEEE Symposium on Security and Privacy – May 2018, Scaling Bitcoin Workshop – October 2018

Scalable and Efficient Distributed Ledger.
CESC – October 2017, Blockchain Unchained – September 2018, ETH Zurich – October 2018,
IBM Research – March 2019

Channels: Horizontal Scaling and Confidentiality on Permissioned Blockchains.
ESORICS – September 2018

Hidden in Plain Sight: Storing and Managing Secrets on a Public Ledger.
Bitcoin Wednesday – November 2018

SERVICE TO COMMUNITY

Lead organizer of Swiss Blockchain Summer School 2017, 2019 (EPFL).
Reviewer: TIFS ’18, EURASIP JIoS ’18, CACM ’19, CBT ’19, SSS ’19, PETS ’19 (external)
Sub-reviewer: CCS ’17, DSN ’18, FC ’19, NSDI ’19

I am also committed to igniting the interest of students for decentralized systems. To this goal, I participated in designing from scratch and lecturing two courses (Information security and privacy, Decentralized systems engineering) at EPFL. Finally, during my PhD studies, I have been lucky to mentor **5** PhD semester projects, **3** MSc thesis, **7** MSc semester projects, **5** Summer interns and **1** BSc project.

LANGUAGES

Greek: Native English: C2 German: B2 French: B1

REFERENCES

Upon request.

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**