Securely Scaling Blockchain Base Layers

Mustafa Al-Bassam

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

December 10, 2020

I, Mustafa Al-Bassam, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

This thesis presents the design, implementation and evaluation of techniques to scale the base layers of decentralised blockchain networks—where transactions are directly posted on the chain. The key challenge is to scale the base layer without sacrificing properties such as decentralisation, security and public verifiability.

It proposes Chainspace, a blockchain sharding system where nodes process and reach consensus on transactions in parallel, thereby scaling block production and increasing on-chain throughput. In order to make the actions of consensus-participating nodes efficiently verifiable despite the increase of on-chain data, a system of fraud and data availability proofs is proposed so that invalid blocks can be efficiently challenged and rejected without the need for all users to download all transactions, thereby scaling block verification.

It then explores blockchain and application design paradigms that enable on-chain scalability on the outset. This is in contrast to sharding, which scales blockchains designed under the traditional state machine replication paradigm where consensus and transaction execution are coupled. LazyLedger, a blockchain design where the consensus layer separated from the execution layer is proposed, where the consensus is only responsible for checking the availability of the data in blocks via data availability proofs. Transactions are instead executed off-chain, eliminating the need for nodes to execute on-chain transactions in order to verify blocks. Finally, as an example of a blockchain use case that does not require an execution layer, Contour, a scalable design for software binary transparency is proposed on top of the existing Bitcoin blockchain, where all software binary records do not need to be posted on-chain.

Impact Statement

Overall, the work in this thesis can be used to inform the design of new and existing projects that implement blockchain networks or applications, in order to improve their scalability and security.

Chainspace, the blockchain sharding system in Chapter 3 was used as part of DECODE (DEcentralised Citizen-owned Data Ecosystems) [128], a European Union project with a digital democracy pilot in Barcelona implementing a decentralised petitions platform. The work in this chapter was also commercialised in a company co-founded by the author of this thesis (`chainspace.io`), which was later acquired by Facebook.

Fraud and data availability proofs in Chapter 4 is being adopted in the specification of Ethereum 2.0 [39], and is used in the whitepaper for Harmony [74], a company implementing a sharded blockchain. The techniques described in this chapter are of general use to any blockchain project to improve the security of light clients. The work on data availability proofs has inspired further work on efficient data availability schemes [156].

LazyLedger in Chapter 5 follows a trend in the blockchain space where the design space of "on-chain data availability, off-chain execution" protocols has received traction recently [38], and contributes one of the first designs following this design paradigm. In the future it may make it economical to build blockchain applications requiring a high level of on-chain data storage. Furthermore, LazyLedger is also being developed as a commercial project (`lazyledger.io`).

Contour, the software binary transparency system in Chapter 6 was developed in the context of law enforcement agencies attempting to bypass end-to-end encryption by compelling software vendors to build versions of their software with backdoors [64]. In 2018, the United Kingdom passed legislation which set out these obligations as part of the Investigatory Powers (Technical Capability) Regulations 2018 [3]. This work (and binary transparency in general) can be used to inform future policy, and also help software vendors increase the security assurances of their users in the context of such legislation.

Acknowledgements

This document would not be possible without my primary supervisor George Danezis and secondary supervisor Sarah Meiklejohn, who have helped me throughout the past few years of research, and whom I am indebted to.

Special thanks to my close collaborators within our research group, Alberto Sonnino and Shehar Bano, who have been the source of many fruitful discussions.

I would also like to thank all of my other co-authors: Sarah Azouvi, Vitalik Buterin, Alexander Hicks, Dave Hrycyszyn, Aggelos Kiayias, William J. Knottenbelt, Eleftherios Kokoris-Kogias, Michał Król, Vasilis Mavroudis, Patrick McCorry, Pedro Moreno-Sanchez, Steven J. Murdoch, Ioannis Psaras, Argyrios Tasiopoulos, Alexei Zamyatin, and Dionysis Zindros.

I would also like to thank Ramsey Khoury, my other co-founder at Chainspace not mentioned above, and all my other former colleagues at Chainspace: Penny Andrews, Andy Bennett, Stuart Chinery, Jérémy Letang, and Lola Oyelayo-Pearson.

As part of this thesis, LazyLedger in Chapter 5 is being commercialised. I would like to thank my co-founders at LazyLedger: John Adler and Ismail Khoffi.

Finally, I would like to thank the Alan Turing Institute for funding this research with a PhD scholarship.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Money is not an invention of the state. It is not the product of a legislative act. Even the sanction of political authority is not necessary for its existence. Certain commodities came to be money quite naturally, as the result of economic relationships that were independent of the power of the state.

Carl Menger

Blockchain-based cryptocurrencies and networks such as Bitcoin [109] and Ethereum [36] are able to operate without a trusted central authority to verify transactions, because nodes in the network verify for themselves that all transactions are valid, and reject blocks that contain invalid transactions. Therefore, the producer of a block (*i.e.,* a miner) does not need to be trusted to only produce blocks with valid transactions, as invalid blocks will be rejected by all correctly functioning nodes in the network.

The public verifiability of blockchains has opened the doors to a new breed of decentralised systems and computation platforms that do not rely on trusted central parties. However, this blessing is also a curse; if every node in the network executes and verifies every transaction, then the transaction throughput of the network is limited by the node with the lowest amount of resources.

Blockchain networks typically adopt a hard limit on the size of blocks [11, 153], to allow nodes with a low amount of resources to participate in the network. While this is positive for the decentralisation of the network as it lowers the cost of running a node, it limits the transaction throughput of the network. During peak times, users pay higher fees as they compete to get their transactions included on the blockchain due to on-chain space being limited. Popular services have stopped accepting Bitcoin payments due to transactions fees rising as high as $20 [115, 81], and Ethereum's popular CryptoKitties smart contract caused the pending transactions backlog to increase six-fold [152].

While increasing the block size limit would increase transaction throughput, this would harm decentralisation as it would increase the cost of running a node to execute and verify transactions. Therefore, for blockchain applications to gain widespread usage, it is necessary to develop blockchain protocols and applications that support greater transaction throughput, without sacrificing the desirable properties of blockchains such as public verifiability and decentralisation.

What is decentralisation?

The word 'decentralisation' can have many meanings and contexts in various systems. In the context of blockchains, we identify two primary contexts for decentralisation:

- **Decentralisation of block production.** This concerns the function of adding new blocks or state to the system.

  Potential questions: How many different entities share the ability to append blocks to the chain? How 'hard' would it be to corrupt a majority of them so that they censor or double-spend transactions? How expensive would it be to bribe them? How many governments would need to co-operate to regulate them?

- **Decentralisation of block verification.** This concerns the function of verifying the validity of blocks and state of the system.

  Potential questions: How expensive is it to verify all the transactions in the blockchain? How many users are doing it in practice? Can users gain an assurance of the validity of blocks in a more efficient way than verifying every transaction individually?

In this thesis, we are focused on the decentralisation of verification, as this thesis proposes the design of scalable transaction execution models. When transaction execution is scaled, transaction verification must also be scaled in order to maintain decentralisation.

1.1 Scope and Purpose

The scope and purpose of this thesis is to develop better blockchain protocols that support greater throughput of transactions posted on the chain. The on-chain (layer 1) protocols of blockchains and their security properties will be discussed. Other aspects of blockchain scalability such cryptographic primitives (other than using them as a pre-requisite), networking protocols (layer 0) or off-chain (layer 2) scalability are out of scope. The primary contribution of this thesis is to design and evaluate new on-chain blockchain protocols that improve the state of the art in terms of scalability and security.

First, we provide background in Chapter 2 into the pre-requisite concepts that this thesis builds upon, including cryptographic building blocks, data structures and transactions. In particular, we focus on Bitcoin [109] and Ethereum [36] as examples of the two most prominent blockchain architectures for payments and smart contracts, respectively.

We make a contribution to scaling blockchains through sharding in Chapter 3, by designing a data model for transaction that enables them to be executed in parallel, and a protocol to facilitate atomic cross-shard transactions (Sharded Byzantine Atomic Commit).

One of key challenges with on-chain scalability is increasing on-chain transaction throughput, but still enabling nodes with low resources to validate the chain (*e.g.,* in all shards) in some way, and reject invalid blocks. In Chapter 4 we thus propose a technique that enables nodes with low resources to run light clients, and receive fraud proofs of invalid blocks from full nodes with more resources. We also contribute a technique–data availability proofs–to allow these light clients to efficiently ensure that the data necessary to generate fraud proofs has been published by block producers.

Data availability proofs are of general interest to verify that an entire file (such as a block) has been published, by only downloading a small sample of that file. In Chapter 5 we use these proofs as a primitive to propose LazyLedger, a new scalable design paradigm for blockchains where the chain is only used as a data availability layer to post ordered messages, but those messages are executed only by end-user clients rather than consensus nodes. The on-chain consensus layer is thus separated from the execution layer, thus simplifying the blockchain to a verifiable log that orders messages, leaving the interpretation of those messages to clients.

As will be shown in Chapter 4, data availability proofs have a scale-out property similar to sharding, where the size of the data that one can prove availability for is proportional to the number of nodes in the network. Therefore by reducing the task of block validation to data availability validation, the LazyLedger system in Chapter 5 achieves scale-out block validation for data availability without sharding the main chain.

Finally, in Chapter 6, we demonstrate how to build a scalable verifiable log on top of the Bitcoin blockchain, for software binary transparency–an application that does not need any execution, but simply an append-only log.

The work in Chapter 5 and Chapter 6 can be thought of as an inversion of each other, as Chapter 5 builds an independent blockchain simplified as a verifiable log, which developers can use to build applications on top of. On the other hand, Chapter 6 builds a scalable verifiable log application that is built on top of the Bitcoin blockchain, instead of a blockchain that natively supports data availability proofs. The goal is to benefit from the existing anti-forking economic security of the Bitcoin blockchain, rather than needing to bootstrap a new smaller network with less security.

1.2 Schedule of Work

1.2.1 Included Work

Parts of this thesis have been published in the following papers:

- Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: 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. The Internet Society (2018), `http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-2_Al-Bassam_paper.pdf`. Included in **Chapter 3**.

- Al-Bassam, M., Meiklejohn, S.: Contour: A practical system for binary transparency. In: García-Alfaro, J., Herrera-Joancomartí, J., Livraga, G., Rios, R. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11025, pp. 94–110. Springer (2018). doi: 10.1007/978-3-030-00305-0_8, `https://doi.org/10.1007/978-3-030-00305-0_8`. Included in **Chapter 6**.

Other parts of this thesis are under submission to conferences or workshops, and/or have been published as pre-prints:

- Al-Bassam, M., Sonnino, A., Buterin, V.: Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. CoRR **abs/1809.09044** (2018), `http://arxiv.org/abs/1809.09044`. Included in **Chapter 4**.

- Al-Bassam, M.: LazyLedger: A distributed data availability ledger with client-side smart contracts. CoRR **abs/1905.09274** (2019), `http://arxiv.org/abs/1905.09274`. Included in **Chapter 5**.

1.2.2 Other Work

As part of my research, the following papers were published that are not included in this thesis:

- Al-Bassam, M.: SCPKI: A smart contract-based PKI and identity system. In: ACM Workshop on Blockchain, Cryptocurrencies and Contracts. pp. 35–40. BCC '17, ACM, New York, NY, USA (2017). doi: 10.1145/3055518.3055530, `https://doi.org/10.1145/3055518.3055530`

- Azouvi, S., Al-Bassam, M., Meiklejohn, S.: Who am I? secure identity registration on distributed ledgers. In: García-Alfaro, J., Navarro-Arribas, G., Hartenstein, H., Herrera-Joancomartí, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10436, pp. 373–389. Springer (2017). doi: 10.1007/978-3-319-67816-0_21, `https://doi.org/10.1007/978-3-319-67816-0_21`

- Sonnino, A., Al-Bassam, M., Bano, S., Meiklejohn, S., Danezis, G.: Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society (2019), `https://www.ndss-symposium.org/ndss-paper/coconut-threshold-issuance-selective-disclosure-credentials-with-applications-to-distributed-ledgers/`

- Król, M., Sonnino, A., Al-Bassam, M., Tasiopoulos, A.G., Psaras, I.: Proof-of-prestige: A useful work reward system for unverifiable tasks. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019. pp. 293–301. IEEE (2019). doi: 10.1109/BLOC.2019.8751406, `https://doi.org/10.1109/BLOC.2019.8751406`

- Bano, S., Sonnino, A., Al-Bassam, M., Azouvi, S., McCorry, P., Meiklejohn, S., Danezis, G.: SoK: Consensus in the age of blockchains. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019. pp. 183–198. ACM (2019). doi: 10.1145/3318041.3355458, `https://doi.org/10.1145/3318041.3355458`

- Sonnino, A., Bano, S., Al-Bassam, M., Danezis, G.: Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In: IEEE European Symposium on Security and Privacy 2020 (2020)

Finally, as part of my research, the following papers are under submission that are not included in this thesis:

- Hicks, A., Mavroudis, V., Al-Bassam, M., Meiklejohn, S., Murdoch, S.J.: VAMS: verifiable auditing of access to confidential data. CoRR **abs/1805.04772** (2018), `http://arxiv.org/abs/1805.04772`
- Al-Bassam, M., Sonnino, A., Król, M., Psaras, I.: Airtnt: Fair exchange payment for outsourced secure enclave computations. CoRR **abs/1805.06411** (2018), `http://arxiv.org/abs/1805.06411`
- Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., Knottenbelt, W.J.: SoK: communication across distributed ledgers. Cryptology ePrint Archive, Report 2019/1128 (2019), `https://eprint.iacr.org/2019/1128`

1.2.3 Work Done in Collaboration

A large part of this work has been conducted in collaboration with other researchers, who are listed in Section 1.2.

In Chapter 3, Alberto Sonnino designed Chainspace's object and smart contract model, whereas I designed the cross-shard transaction protocol (Sharded Byzantine Atomic Commit) and led the implementation of the smart contracts framework and evaluation of the system.

In Chapter 4, Alberto Sonnino designed the security theorems and implemented the fraud proofs prototype, Vitalik Buterin proposed the idea of using 2D erasure coding for data availability proofs, and I designed the overall fraud and data availability proofs system, and implemented and measured the performance of data availability proofs.

Chapter 5 is sole work that was not done in collaboration.

In Chapter 6, I designed, implemented and measured the performance of the Contour system. My secondary supervisor Sarah Meiklejohn helped with the formalisation of the system design.

Chapter 2

Background

The only true wisdom is in knowing you know nothing.

Socrates

2.1 Cryptographic Primitives

This section presents the basic cryptographic building blocks that underpin cryptocurrencies and blockchain protocols.

2.1.1 Hash Functions

A cryptographic hash function is a publicly known function h that takes in an arbitrary sized input x, and returns an output of fixed size $H(x)$. There are three desirable properties for a cryptographic hash function:

- **Pre-image resistance.** Given a hash y, it is computationally hard to find any x such that $H(x) = y$.

- **Weak collision resistance.** Given an input x_1, it is computationally hard to find a different input x_2 such that $H(x_1) = H(x_2)$.

- **Strong collision resistance.** It is computationally hard to find any x_1 and x_2 such that $H(x_1) = H(x_2)$. This implies weak collision resistance.

In cryptocurrencies, common hash functions include SHA-256 which is used by Bitcoin [11], and SHA-3 which is used by Ethereum [153].

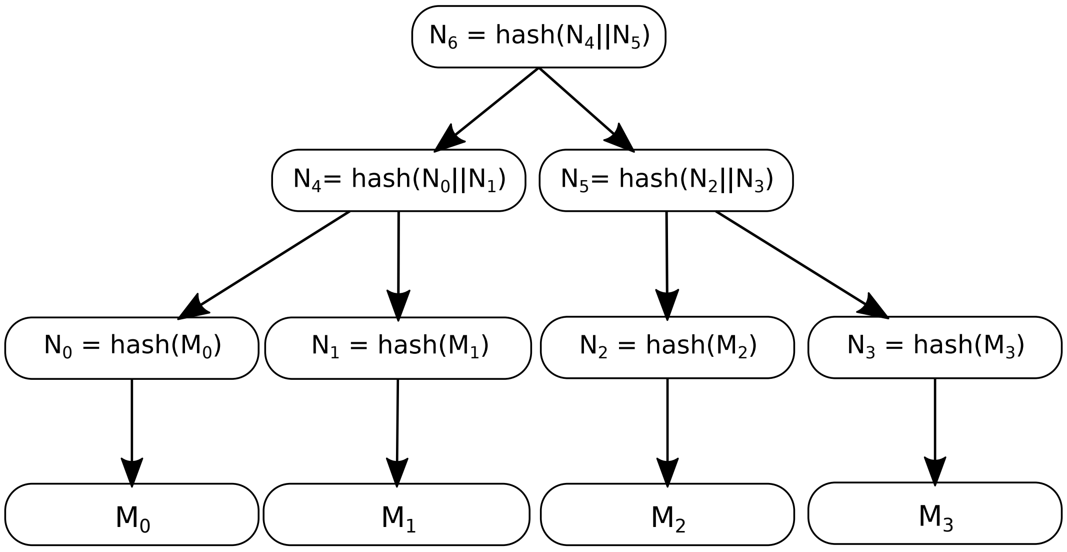


Figure 2.1: Illustration of an example Merkle tree where the items being committed to are M_0, M_1, M_2 and M_3. The leaf nodes are N_0, N_1, N_2 and N_3, the intermediate nodes are N_4 and N_5, and the root of the tree is N_6.

2.1.2 Merkle Trees

A Merkle tree [106] is a binary tree where every non-leaf node is labelled with the cryptographic hash of the concatenation of its children nodes. The root of a Merkle tree can thus be shown to be a commitment to all of the items in its leaf nodes. Figure 2.1 illustrates an example Merkle tree.

This allows for Merkle proofs which, given the root of some Merkle tree, are proofs that a leaf is a part of the tree committed to by the root (*i.e.,* a proof of set membership). A Merkle proof for some leaf consists of all of the sibling nodes of the ancestor nodes of that leaf, up to the root of the tree, as illustrated in Figure 2.2. This allows a verifier to recompute the ancestor nodes of the leaf and the root, to verify that the Merkle proof is valid and matches the committed root of the tree. The size and verification time of a Merkle proof for a tree with n leaves is $O(\log(n))$, as it is a binary tree.

Sparse Merkle trees

A sparse Merkle tree [93, 51] is a Merkle tree with n leaves where n is extremely large (*e.g.,* $n = 2^{256}$), but where almost all of the nodes have the same default value

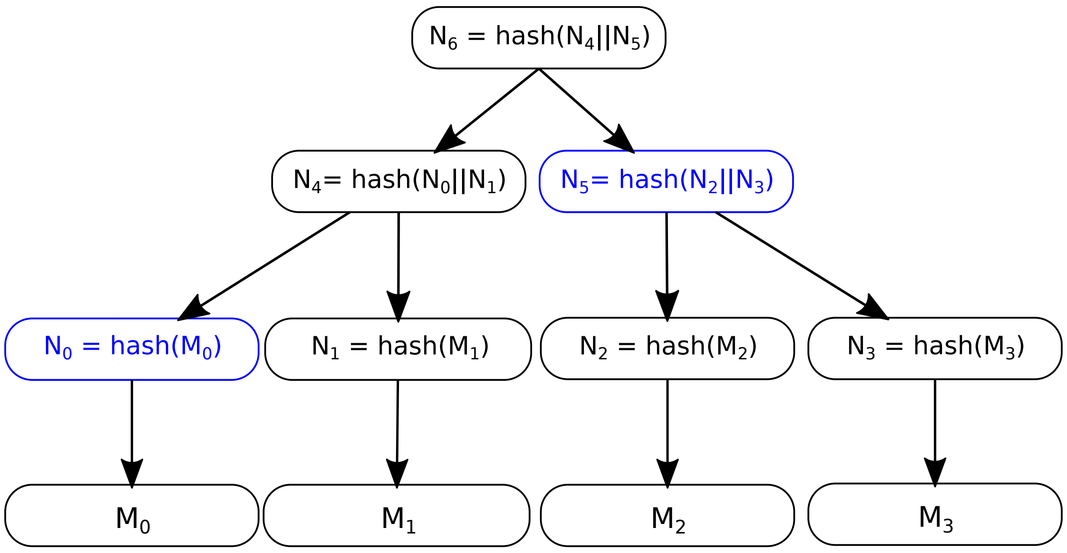


Figure 2.2: A Merkle proof for M_1, where the blue nodes are included in the proof (in addition to M_1 itself and the root hash N_6).

(*e.g.,* 0). If k nodes are non-zero, then at each intermediate level of the tree there will be a maximum of k non-zero values, and all other values will be the same default value for that level: 0 at the bottom level, $L_1 = H(0||0)$ at the first intermediate level, $L_2 = H(L_1||L_1)$ at the second intermediate level, and so on, where H is a hash function. Hence despite the exponentially large number of nodes in the tree, the root of the tree can be calculated in $O(k \times \log(n))$ time.

A sparse Merkle tree allows for commitments to key-value maps, where values can be accessed, updated, inserted or deleted trivially in $O(\log(n))$ time. Merkle proofs of specific key-values entries are of size $\log(n)$ if constructed naively but can be compressed to size $\log(k)$ as intermediate nodes whose sibling have the default value do not need to explicitly be shown.

In a sparse Merkle tree, the $H(k)$th leaf node represents the value of the key k. Figure 2.3 illustrates a sparse Merkle tree.

Their are also optimisations for sparse Merkle trees to allow values to be accessed or updated in $O(\log(k))$ time rather than $O(\log(n))$, with respect to the number of hash operations that need to performed. This can be achieved in one of two ways:

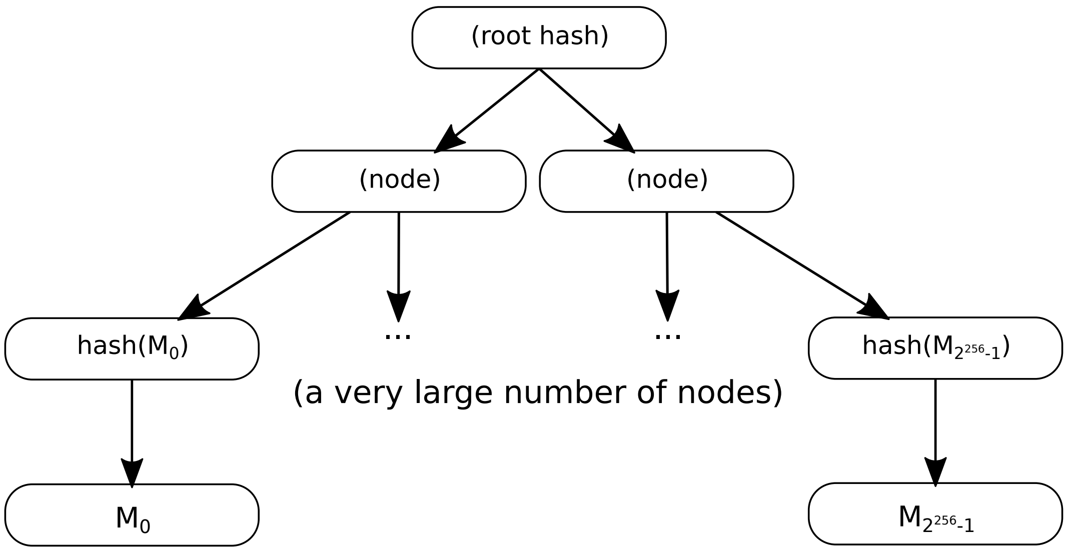


Figure 2.3: Illustration of a sparse Merkle tree with 2^{256} items.

1. Use a 'wrapper' hash function $W(l||r)$ that calls a real hash function H, where W is trivial to compute if l or r have default values [40]. Thus, H only needs to be called $log(k)$ times rather than $log(n)$ times, meaning that there are only $log(k)$ 'non-trivial' computations.

2. Simply replace subtrees consisting of only one non-default leaf, with that leaf. Additionally, subtrees consisting of only default leaves are replaced with a placeholder value [10].

There are other types of Merkelised trees that support key-value maps, most notably Patricia trees used by Ethereum and Ripple [153, 132]. However in this thesis we focus on sparse Merkle trees as our primitives for key-value maps due to their greater simplicity and equivalent performance (with optimisations).

2.1.3 Public-Key Cryptography and Signatures

Public-key cryptography uses pairs of keys: a private key used to decrypt or sign messages, and a corresponding public key used to encrypt messages to the private key holder and verify signatures made by the private key holder. In cryptocurrencies, public-key cryptography is primarily used for digital signatures.

Given a key-pair $(K_{\mathsf{pk}}, K_{\mathsf{sk}})$ where K_{pk} is a public key and K_{sk} is its corresponding private (secret) key, and a message x, there is a function $\mathsf{sign}(K_{\mathsf{sk}}, x)$ that returns

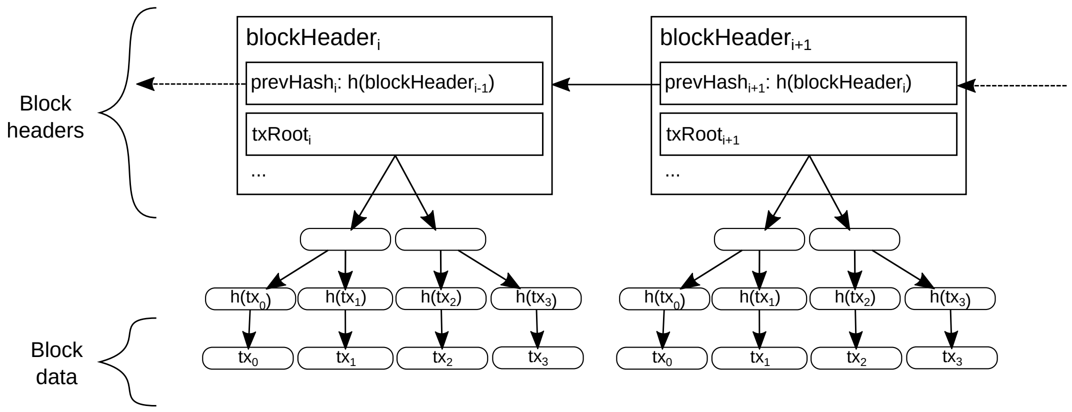


Figure 2.4: Illustration of the core elements of a blockchain.

a signature for the message x, and a boolean verification function $\mathsf{verify}(K_{\mathsf{pk}}, m, s)$ that returns true if the signature s for a message m by public key K_{pk} is correct, so that $\mathsf{verify}(K_{\mathsf{pk}}, x, \mathsf{sign}(K_{\mathsf{sk}}, x))$ is true.

There are many different algorithms for public-key cryptography systems. The most common system used for cryptocurrencies is the Elliptic Curve Digital Signature Algorithm, which is used by Bitcoin and Ethereum [11, 153].

2.2 Blockchain Data Structures

The data structure of a blockchain is relatively simple. It consists of a sequence of data elements–called blocks–such that each block contains the cryptographic hash of the previous block.

As illustrated in Figure 2.4, each block consists of two core components: the block header, and the block data. The header is relatively short piece of data that contains the 'meta-data' of the block. At minimum, a block header contains the hash of the previous block header in the chain, thus enabling the chaining property, and the Merkle root of all the transactions in the block. The block data is the raw transactions in the block, *i.e.,* all the leafs of the Merkle tree.

Given a current block header, the blockchain is append-only: no transactions or data in the previous blocks in the chain can be modified without changing the hashes of their block headers, and thus all of the future block headers including the current block header. Instead, data can only be appended to the chain by creating a

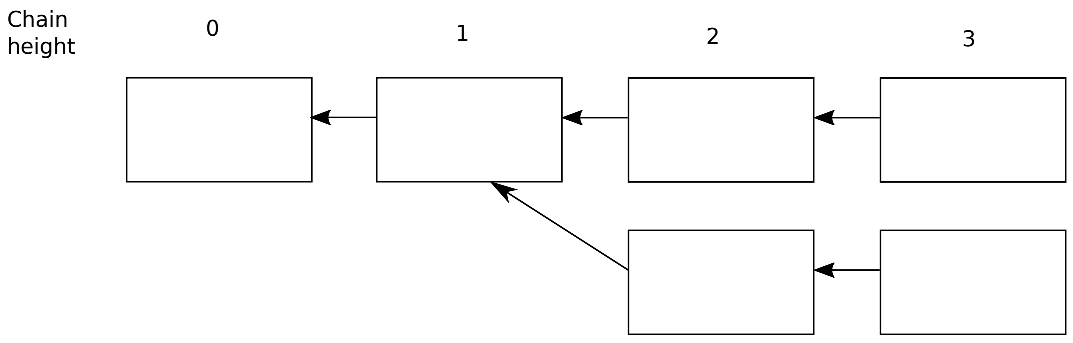


Figure 2.5: A blockchain fork.

new block with a header that references the current bock header.

The block header can also contain other data that is specific to the blockchain or consensus protocol, as we shall see later on in Section 2.3.

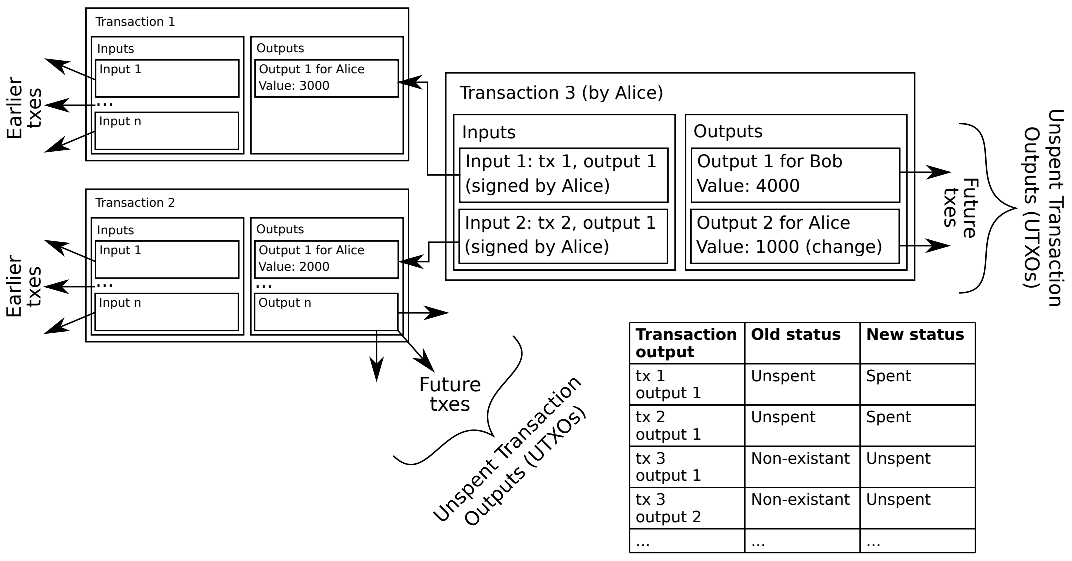
2.2.1 Genesis Block

The very first block in the chain is known as the ‘genesis block’, and is typically hard-coded into the software. This requires a degree of trust in the creator of the genesis block (trusted setup), because if the block was created a long time before the software was released, the creator would have a significant head-start in crafting an attack to ‘fork’ the chain in the future (see Section 2.2.2), depending on the consensus protocol (see Section 2.3.2).

For this reason, the creator of Bitcoin embedded the message ‘The Times 03/Jan/2009 Chancellor on brink of second bailout for banks’ into the genesis block [110], to prove that the block was created after the date specified in the referenced newspaper headline.

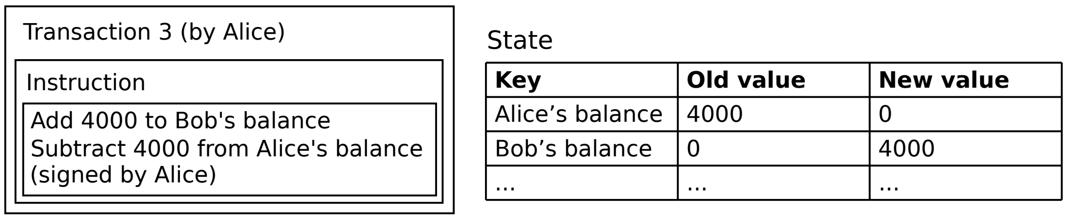
2.2.2 Forks

As illustrated in Figure 2.5, a blockchain may contain a ‘fork’. A fork occurs when a block header is succeeded by multiple block headers that reference it as the previous header. Blockchain systems employ consensus algorithms (to be discussed in Section 2.3.2) to resolve forks and to ensure that there is only one canonical chain that is considered the true chain by nodes in the network.



Transaction output	Old status	New status
tx 1 output 1	Unspent	Spent
tx 2 output 1	Unspent	Spent
tx 3 output 1	Non-existant	Unspent
tx 3 output 2	Non-existant	Unspent
...	...	...

Figure 2.6: High-level overview of UTXO-based transactions.



Key	Old value	New value
Alice’s balance	4000	0
Bob’s balance	0	4000
...	...	...

Figure 2.7: High-level overview of account-based transactions.

2.2.3 Transactions

Addresses

When Alice wishes to send money to Bob, Alice generates a transaction that sends money to Bob’s public key, spending money controlled by Alice’s public key. The identifiers of these public keys are known as *addresses*. For example, in Bitcoin, an address is the Base58 encoded version of a hashed representation of the public key. To prove that Alice is authorised to spend the money controlled by Alice’s public key, the transaction is signed by Alice’s private key.

There are two main types of blockchain-based transaction models, which are discussed below.

Unspent Transaction Output (UTXO)-Based Transactions

In this transaction model, every transaction has multiple monetary *inputs* and *outputs*. Figure 2.6 provides an overview of UTXO-based transactions.

Outputs can specify which addresses are being paid by the transaction, and how much. Specifically, outputs are scripts that specify the conditions necessary for that output to be spent, or “unlocked”. For example in Bitcoin, outputs can be a script that returns true upon the production of a valid cryptographic signature associated with the address specified in the output, that wishes to claim that output. This is known as a Pay-to-PubkeyHash (P2PKH) transaction, as the sender is sending a payment to the hash of a public key (an address).

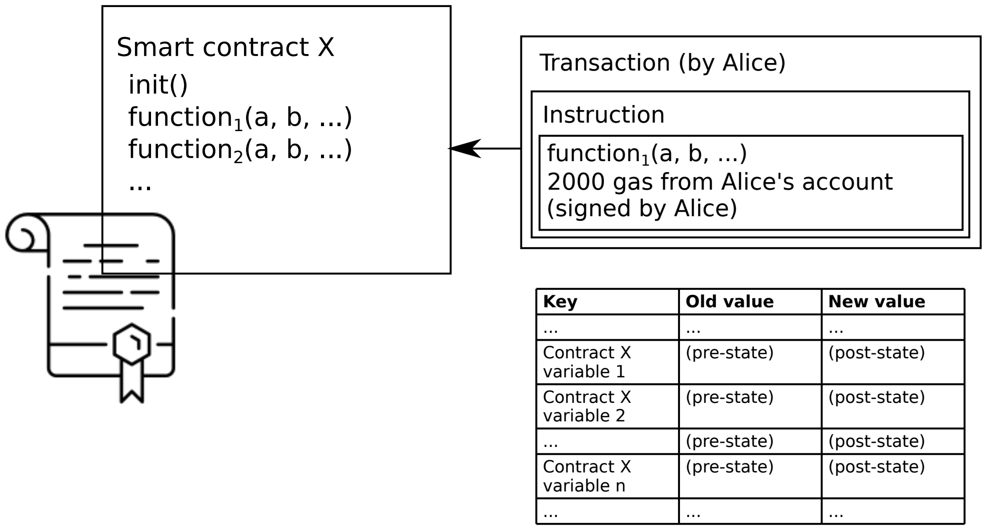
When a user wishes to spend money sent to their address, they must create a transaction whose inputs are references to outputs from previous transactions, and a “script signature” for each input that specifies a value that causes the input script to return true, such as a valid cryptographic signature. Each output can only be spent once, and unspent outputs are known as Unspent Transaction Outputs, or UTXOs for short.

This enables nodes in the network to not keep track of the balances of all the addresses in the system. They simply keep track of all of the unspent transaction outputs in order to know which public keys are entitled to which funds, and to validate new transactions. The state of the ledger at a given point in time can thus be expressed as the set of unspent transaction outputs.

As unspent outputs can only be spent once, all of the monetary value of that output must be consumed by the transaction spending the output at once. This means that if Alice is attempting to spend an output to pay Bob, but the value she wants to pay Bob is less than the value of the output, then she must create an additional “change” output in the transaction that sends the excess money back to an address she controls.

Account-Based Transactions

Unlike the UTXO-based transaction model, the account-based transaction model does keep track of balances of addresses, which are known as “accounts”. Figure 2.7



Key	Old value	New value
...	...	...
Contract X variable 1	(pre-state)	(post-state)
Contract X variable 2	(pre-state)	(post-state)
...	(pre-state)	(post-state)
Contract X variable n	(pre-state)	(post-state)
...	...	...

Figure 2.8: High-level overview of account-based smart contracts.

provides an overview of account-based transactions.

Transactions do not have inputs or outputs, but simply specify from and to address to transfer money between. Nodes in the network need to keep track of the balance of all addresses in order to be able to validate new transactions. The state of the ledger at a given point in time can thus be expressed as a key-value map, where the keys are accounts and the values are balances.

Although this model is simpler than the UTXO-based model, its disadvantage is that it is more difficult to apply parallelisation to, as a balance can only be safely updated sequentially. On the other hand, UTXOs are independent of each other and thus multiple UTXOs for the same address can be created and spent in parallel.

2.2.4 Smart Contracts

The language that Bitcoin transaction outputs are written in, called **Script**, is a simple stack-based language with a very limited set of opcodes [11]. It can be used to add simple conditions to transaction outputs, such as freezing funds until a time in the future, or creating M-of-N multi-signature transactions that require signatures from M out of N parties to spend an output.

Since then, blockchain platforms with more advanced scripting languages have emerged, the first and most popular of which being Ethereum [36]. These platforms support the use of "smart contracts", scripts which can be uploaded to the blockchain, that define functions that users can call by sending transactions, to modify the smart contract's state on the blockchain. In this model, the script is therefore not defined by the transaction or its outputs, but by a pre-defined smart contract. When a new contract is uploaded, its state is typically initialised with an initialisation function defined by the smart contract.

Ethereum smart contracts are written in specialised high-level languages that are compiled to Ethereum Virtual Machine (EVM) code [153], a limited but Turing-complete [147] execution environment. As illustrated in Figure 2.8, smart contract environments typically employ an account-based model, where the state of smart contracts are stored in a key-value store that can be modified by the smart contract. However in Chapter 3 we will discuss a UTXO-based smart contract environment optimised for parallel transaction execution.

A smart contract execution environment must have several properties that differentiate it from standard programming environments, to make it suitable within the adversarial setting of a decentralised platform. Such an execution environment must at minimum (*i*) be compartmentalised from the system in which it is run on (*e.g.,* it cannot make system calls) as code is run from untrusted sources, (*ii*) be deterministic in its execution, regardless of where it is run, so that all nodes verifying transactions can arrive to the same result, and (*iii*) as we shall see in Section 2.3.4, have a deterministic way to measure precisely the amount of computational resources its programs consume, regardless of where it is run and how long it takes to run.

For this reason, standard programming languages such as Python are not suitable for smart contracts, hence the creation of the Ethereum Virtual Machine. However more recently, there has been work on using WebAssembly for smart contracts [62]. Given that it is designed to run in web browsers and run untrusted code from websites, it is suitable for smart contracts programming with some modifications to its standard library to make it deterministic.

Publicly callable smart contract functions can also be called by other smart contracts, thus allowing developers to compose other smart contracts, or use code in other contracts as libraries. Smart contracts may have their own balances, so that they can act like an address with the ability to send and receive funds.

2.2.5 State Commitments

Blockchains with newer designs include an additional piece of data in block headers known as a 'state commitment'. This is a cryptographic value that commits to the state of the blockchain at the point the block was created, and allows clients to efficiently verify that some value is a part of the state, upon a prover generating and presenting a proof of this with the commitment.

Typically, this state commitment is the root of a Merkelized tree that supports non-membership proofs (*i.e.,* proof that some key or element is not in the tree) such as a sparse Merkle tree (as described in Section 2.1.2).

A state commitment based on a Merkelized tree that supports a key-value map, can be created with both a UTXO-based and an account-based blockchain:

- **UTXO-based.** The keys in the map are transaction output identifiers *e.g.,* $\mathsf{H}(\mathsf{H}(\mathsf{txData})||i)$ where txData is the raw transaction data and i is the index of the output being referred to in txData. The value of each key is the state of each transaction output identifier: either *unspent* (1) or *nonexistent* (0, the default value).

- **Account-based.** This is already a key-value map, where the key is the account or storage variable, and the value is the balance of the account or the value of the variable.

In the case of a UTXO-based blockchain, a state commitment can also simply be a standard Merkle tree containing the set of all UTXOs. However, it would not be possible to prove that an UTXO does not exist as a standard Merkle tree does not support non-membership proofs.

It is also possible to use cryptographic accumulators such as RSA accumulators for state commitments [31], but this is outside the scope of this thesis.

2.3 Blockchain Peer-to-Peer Network

In Section 2.2, we discussed the data structure of a blockchain, and the mechanics of transactions and smart contracts. In this section, we will discuss how these pieces fit together within a peer-to-peer network to construct a live system.

In order to implement a blockchain within a network, there are two key elements to consider:

- *Who* is permitted to participate in the process of appending blocks to the blockchain. This is typically a group of nodes. This is known as Sybil-resistance [56], and will be discussed in Section 2.3.1.
- *How* the group of nodes that are permitted to append blocks to the blockchain agree with each other which block should be added, or whose turn it is to propose a block. This is known as consensus or *Byzantine fault tolerance*, and will be discussed in Section 2.3.2.

2.3.1 Sybil-resistance

A Sybil attack [56] is an attack that occurs in peer-to-peer systems where a node in the network operates under multiple identities, thus appearing to be multiple nodes. The Sybil attacker can launch many identities, or "Sybils", to gain the majority of the influence in the network. In the context of blockchain networks, this means that the Sybil attacker controls the majority of the group of nodes that are responsible for appending blocks to the chain, and thus may fork the chain to double spend a transaction.

A key challenge of building decentralised blockchains with no centralised trusted parties is to develop a mechanism to allow nodes who participate in the consensus to be chosen in a decentralised way, such that it would be difficult for any single party to conduct a Sybil attack. This section discusses the two most prominent decentralised Sybil-resistant node selection schemes, proof-of-work and proof-of-stake. First we discuss permissioned blockchains, which simply use a centralised trusted party to select nodes rather than a decentralised Sybil-resistant mechanism.

Permissioned

Rather than using a decentralised Sybil-resistant mechanism that allows any party to participate in the production of blocks, some blockchain designs require the set of consensus-participating nodes to be selected by some centralised party or group of parties. This is known as a permissioned or consortium blockchain, and is often used in an enterprise setting where the blockchain is run and governed by a consortium of companies that do not wish to open participation to external parties.

Libra [10] is an example a permissioned blockchain that is operated by a group of one hundred nodes run by different companies and organisations, who are members of a legal entity known as the Libra Association. Hyperledger [29] is a popular software project which provides a series of tools and frameworks designed to enable enterprises to run their own permissioned blockchains.

Proof-of-work

Proof-of-work is the Sybil-resistance scheme used by Bitcoin and its derivatives. The idea of proof-of-work was first presented by Dwork and Naor in 1993 as a technique for combatting spam mail, by requiring the email sender to compute the solution to a mathematical puzzle to prove that some computational work was performed [58].

Proof-of-work was independently proposed in 1997 for Hashcash by Back, another system for fighting spam [18]. In Hashcash, the computational puzzle is finding a SHA-1 hash of a header including the email recipient's address and current date, such that the hash contains at least 20 bits of leading zeros. As the hashing algorithm is pre-image resistant, the puzzle can be solved only by sampling random nonces in the header until the resulting hash meets the leading zeros requirement. Testing these guesses require a significant amount of computational work, so a valid hash is considered to be a proof-of-work.

Bitcoin's proof-of-work mechanism is derived from Hashcash [18]. It replaces Hashcash's SHA-1 hashing with two successive SHA-2 hashes, and requires valid hashes to have a value below a target integer value t. The difficulty of the puzzle is therefore adjustable: decreasing t increases the number of guesses (and thus work) required to generate a valid hash. The nodes that generate hashes are called *miners* and the process is referred to as *mining*. Miners calculate hashes of candidate blocks of transactions to be added to the blockchain, and are rewarded with new coins if they find a valid block. The value t is reset by the network every 2016 blocks such that miners are successful (and can append a block to the blockchain) on average every 10 minutes (also called the *inter-block interval*).

This means that the more computing power a miner has, the more likely they are to successfully mine a block. Thus each miner's share of 'voting power' to decide the next blocks in the system is proportional to how much computational resources they have.

In order to conduct a Sybil attack in the network, the attacker needs the majority of the computational resources in the network [109], although more recent strategies show that such an attack can be performed with only 25% of resources [63].

Proof-of-stake

Due to concerns that proof-of-work requires a significant amount of electrical energy to secure the network, proof-of-stake has emerged as a more energy-efficient alternative.

In proof-of-stake, the 'voting power' or likelihood of nodes deciding the block is proportional to how many coins they hold in the system (*i.e.,* their stake in the system), rather than computational resources.

Compared to proof-of-work, proof-of-stake protocols are subject to several key attacks that need to be address [20]:

- **Nothing-at-stake attack.** Unlike proof-of-work, there is no cost to creating a block in proof-of-stake. Thus block producers in proof-of-stake ('stakers') have the incentive to attempt to extend every fork, in the hope that they will get their block included in the right fork that reaches consensus. Mitigations include introducing penalty fines ('stake slashing') for stakers that extend the wrong fork [41].

- **Grinding attack.** In proof-of-stake protocols, a source of randomness is needed in order to randomly select a staker out of the set of stakers to produce the next block. In a grinding attack, a staker attempts to influence the random number generation algorithm in its own favour, for example if the algorithm's source of randomness (seed) includes data supplied by the staker such as block header data. This is mitigated by using a source of randomness that cannot easily be biased or manipulated [83, 17].

- **Long-range attack.** In this attack, an attacker may purchase private keys from stakers that had stored significant coins in the past, but not presently. If the attacker has sufficient private keys such that they control a majority of coins in the past, they can use these keys to fork the chain from a point in the past an re-write the entire history of the chain. This can be mitigated by creating checkpoints of the chain that finalise the chain at every checkpoint [41]. Unfortunately, this means that if a node goes offline for a long period of time and miss the creation of the checkpoint, they must ask another trusted node to tell them which is the correct checkpoint (in the event that there are multiple adversarial checkpoints).

Others

Although proof-of-work and proof-of-stake are the most prominent Sybil-resistant mechanisms, other less prominent mechanisms have been proposed. This includes proof-of-burn, where block producers must destroy coins to generate blocks [85]. Another is proof-of-coin-age, a version of proof-of-stake where the power of block producers is proportional to how much coins they hold weighted by the time since they were last moved [116].

2.3.2 Consensus and Byzantine Fault Tolerance

Once it is decided who is a part of the consensus group, the group must come to consensus on the blocks to be attended to the chain. In this section we categorise consensus into three main categories: centralised (*i.e.,* a single trusted third party), Nakamoto consensus (as introduced by Bitcoin [109], and tradition Byzantine fault tolerance algorithms.

Centralised/Verifiable Logs

Some system designs may opt to use a blockchain data structure for its append-only property only, but not for decentralisation. This useful in a use case where the goal is to write all of the actions of a centralised service in a tamper-resistant log, so that evidence of misbehaviour cannot be deleted from the log. Such systems have been proposed in the context of tamper-resistant operating system logs [50] and TLS certificate transparency [94].

In such systems, the consensus algorithm is typically simply a centralised log server signing block headers [50, 94], and thus the consensus group consists of one party with no Byzantine fault tolerance. Likewise, there is no Sybil-resistant mechanism as the consensus group consists of a known single party. To prevent the log server from forking the chain and presenting different views of the log to clients (equivocation), a gossiping protocol can be used so that clients can share versions of logs with each other to detect equivocation [113].

In Chapter 6 we will explore how in case of software binary transparency, gossiping does not work and it is better to use Bitcoin as a tamper-resistant verifiable log. This borrows from the economic security of Bitcoin to guarantee tamper-resistance.

Nakamoto Consensus and Longest-chain Rule

Nakamoto consensus was proposed in the Bitcoin whitepaper by Satoshi Nakamoto [109]. It is fairly simple: miners mine blocks with proof-of-work, and in the event multiple blocks are mined at the same height (a fork), the chain with the most accumulated proof-of-work is the correct one. This is (counter-intuitively) known as the longest-chain rule. When applied to proof-of-stake, this would be the chain with the most accumulated proof-of-stake. A fork may occur due to network latency issues; a miner may for example produce a block at the same height of another block that was produced seconds earlier, because the miner did not see that block in time.

Nakamoto consensus is very efficient; it requires only an $O(1)$ messaging complexity for a node to verify that a block has consensus, as it simply needs to download a block header (whose size is independent of the number of nodes in the network) and check that it is in the chain with the most accumulated proof-of-work.

The main disadvantage of Nakamoto consensus that it has a slow finality time; it takes a long time for nodes to be reasonably sure that a block is really part of the chain and will not be orphaned in a fork. The typical recommendation for Bitcoin is to wait for 6 blocks (an hour) for a transaction to be considered confirmed [11].

Traditional Byzantine Fault Tolerance

'Traditional' Byzantine fault tolerance (BFT) consensus algorithms have been proposed since at least the 1980s. In such algorithms, $2f+1$ honest nodes are needed to tolerate f dishonest (or Byzantine) nodes [92, 118].

Unlike Nakamoto consensus, these systems have fast finality because they explicitly require an honest $2f+1$ majority of $3f+1$ participants to sign every message (which could *e.g.,* a block) that consensus is to be reached on. Thus 'forking' is not possible if $2f+1$ participants are honest, as they cannot accidentally approve consensus on two conflicting messages (*e.g.,* two blocks of the same height) due to *e.g.,* network latency issues.

The disadvantage of traditional BFT algorithms is their high messaging complexity; the most prominent BFT protocol, Practical Byzantine Fault Tolerance (PBFT) [43] has a worse-case $O(N^4)$ messaging complexity [52].

More recently, there have been a large variety of hybrid 'protocols' that blur the boundaries between Nakamoto consensus and traditional BFT consensus, to achieve the best of both worlds in finality and efficiency [101, 52, 41, 35]. We do not discuss them here further, as this thesis does not focus on the design of consensus protocols, but uses them as a building block.

2.3.3 Block Validity and Network Nodes

In addition to consensus rules, blockchains also typically have a set of on-chain transaction validity rules that dictate which transactions are valid. Thus blocks that contain invalid transactions will never be accepted by the consensus algorthim and should in fact always be rejected.

Full nodes (also known as ‘fully validating nodes’) are nodes which download both the block headers as well as the list of transactions, verifying that all transactions are valid according to some transaction validity rules. This is necessary in order to know which blocks have been accepted by the consensus algorithm.

There are also ‘light’ clients which only download block headers, and assume that the list of transactions are valid according to the transaction validity rules. These nodes verify blocks against the consensus rules, but not the transaction validity rules, and thus assume that the consensus is honest in that they only included valid transactions. They therefore do not fully execute the consensus algorithm to know which blocks are accepted, and may end up in a situation where they accept blocks that contain invalid transactions, that full nodes have rejected.

Nodes with insufficient resources to run a full node, such as a mobile phone, can run a light client. Light clients can accept payments by verifying that transactions have been included in the chain, by requesting from full nodes Merkle proofs of transactions being included in blocks.

The difference between these node types is at the crux of the blockchain scalability challenge: if on-chain transaction capacity is increased, the amount of resources required to run a full node could increase, thus requiring more people to run light clients that make an additional honest-majority assumption for block validity. This will be discussed in greater depth in Chapter 4.

Gossiping and Eclipse Attacks

Typically, blocks are distributed to nodes via a peer-to-peer gossiping network [34]. An attacker that is able to control which nodes a node connects to can conduct an ‘eclipse attack’ [75], where the node is made to connect to only nodes that the attack controls. The attacker can thus control the node’s view of the network, and control which blocks it sees. This can cause the node to be on a different fork of the blockchain than the rest of the network that is not under an eclipse attack. We explore such attacks in more detail in Chapter 6.

2.3.4 Transaction Fees

Users can choose to optionally include a fee in their transaction, paid to any block producer that includes their transaction in a block. Block producers may only decide to include transactions with the highest fee if block space is limited, thus creating a fee market. The higher the transaction fee, the more likely (and thus the more quickly) it is to be included in a block.

In smart contract platforms, executing a function in a smart contract has a “gas” cost associated with it, depending on how many and which operations the function performs [153]. The more resource intensive a transaction is, the more the user would have to pay. Different operations may have different gas costs depending on how much CPU, RAM or storage space they require. This gas cost is paid by the transaction creator in cryptocurrency tokens, and the transaction creator can choose how much cryptocurrency per gas to include, but this will affect the speed in which the transaction is included in a block, akin to transaction fees described earlier.

If the transaction is being executed in a Turing-complete execution environment, then the transaction creator includes a maximum gas fee, as due to the halting problem [147] it may not be possible to determine the gas cost of the execution beforehand, and the execution may consume differing amount of gas depending on the state of the contract at the point of execution.

Chapter 3

Chainspace: A Sharded Smart Contracts Platform

Whether to concentrate or to divide your troops, must be decided by circumstances.

Sun Tzu

3.1 Introduction and Motivation

In blockchain networks such as Bitcoin and Ethereum, consensus-participating block producers download and process (*i.e.,* validate) all on-chain transactions to propose blocks. The throughput of the chain is therefore bottlenecked by the block producer with the lowest computational and network resources that ought to be supported by the network.

By 'sharding' the network into multiple chains with different consensus groups that produce blocks, different block producers can process different transactions in parallel, rather than every transaction. As the transaction load is then distributed, on-chain throughput can be increased.

Two key challenges in designing a sharded blockchain system are (*i*) designing a protocol for transactions to atomically access and modify state across shards and (*ii*) supporting a transaction model that allows smart contracts to be designed that can be easily parallelised across shards.

The atomic cross-shard transaction problem can be exemplified with the train and hotel problem [54]. Suppose that a user wants to make a transaction to book a train seat and hotel room atomically, so that either both bookings succeed or neither do. If the state for the train seat and hotel room is stored on different shards, then the user could end up in a situation where the transaction succeeds on one shard but fails on the other, so the user will end up with only either a train seat or hotel room. To prevent this, a protocol to facilitate atomic transactions between shards is required.

In this chapter we introduce Chainspace, a distributed ledger design for smart contracts that makes use of sharding to allow nodes to process on-chain transactions in parallel. A primary design goal of Chainspace is to allow for atomic cross-shard transactions. A modest test-bed of 60 nodes achieves 350 transactions per second, as compared with a peak rate of less than 7 transactions per second for Bitcoin over around 5,400 full nodes [49].

The chapter makes the following contributions:

- It presents Chainspace, a system that can scale arbitrarily as the number of nodes increase, tolerates Byzantine failures, and can be fully and publicly audited.

- It introduces a UTXO-like data model for smart contracts based on atoms called ‘objects’, which allows for transaction processing to be parallelised across shards.

- It presents a novel distributed atomic commit protocol, called S-BAC, for sharding generic smart contract transactions across multiple Byzantine nodes, and correctly coordinating those nodes to ensure safety, liveness and security properties.

- It provides a full implementation and evaluates the performance of the Byzantine distributed commit protocol, S-BAC, on a real distributed set of nodes and under varying transaction loads.

3.2 Related Work

In this section we provide a view of related sharding proposals to Chainspace. For a systematisation of knowledge of sharding on blockchains, we refer the reader to SoK: Sharding on Blockchains by Wang *et al.* [149] and a formalisation of blockchain sharding protocols by Avarikioti *et al.* [14].

3.2.1 OmniLedger

The most comparable system to Chainspace is OmniLedger [87]—that was developed concurrently—and provides a scalable distributed ledger for a cryptocurrency, but it is not designed for generic smart contracts by default. OmniLedger assigns nodes (selected using a Sybil-attack resistant mechanism) into shards among which state, representing coins, is split. The node-to-shard assignment is done every epoch using a bias-resistant decentralized randomness protocol [142] to prevent an adversary from compromising individual shards. A block-DAG (Directed Acyclic Graph) structure is maintained in each shard rather than a single blockchain, effectively creating multiple blockchains in which consensus of transactions can take place in parallel. Nodes within shards reach consensus through the PBFT protocol [43] with ByzCoin [86]'s modifications that enable $O(n)$ messaging complexity. In contrast, Chainspace uses BFT-SMART 's PBFT implementation [141] as a black box, and inherits its $O(n^2)$ messaging complexity—however, BFT-SMART can be replaced with any improved PBFT variant without breaking any security assumptions.

Similar to Chainspace, OmniLedger uses an atomic commit protocol to process transactions across shards. However, it uses a different client-driven approach to achieve it, called Atomix. To commit a transaction, the client first sends the transaction to the network. The leader of each shard that is responsible for the transaction inputs (input shard) validates the transaction and returns a proof-of-acceptance (or proof-of-rejection) to the client, and inputs are locked. To unlock those inputs, the client sends proof-of-accepts to the output shards, whose leaders add the transaction to the next block to be appended to the blockchain. In case the transaction fails the validation test, the client can send proof-of-rejection to the input shards to roll back the transaction and unlock the inputs.

3.2.2 Ethereum 2.0

Ethereum 2.0 is a proposed upgrade to Ethereum to deploy sharding. It allows for cross-shard transactions through a mechanism known as ‘cross-shard yanking’ [37], which is a form of a mutual exclusion (mutexes) [55].

Mutexes are a property of computer program threads that run concurrently that access a shared resource, where it is a requirement that only one thread can enter its ‘critical section’ at a time. The critical section of a thread is the part of a thread that accesses a shared resource, where concurrent access to that resource can lead to erroneous behaviour. Therefore, mutexes ensure that only one thread at a time is accessing a shared resource.

In the context of sharding, the threads are shards, and in Ethereum, the shared resources are smart contracts. The goal is to ensure that a contract is accessed by only one shard at a time.

Suppose Alice wants to yank a contract X from shard A to B. Cross-shard yanking works as follows [37]:

- Alice creates a transaction on shard A that issues a $\mathsf{yank}(X,B)$ command to yank contract X from shard A to shard B.

- If they yank command is successful, shard A creates a ‘receipt’ that contains the state of contract X, and the target shard B. This receipt is stored in a block shard A produces, such that Alice can generate a Merkle proof of the receipt being included in the block.

- Alice sends the receipt to shard B, which then imports and creates the smart contract and its state in shard B.

If Alice wants to perform a transaction that involves two contracts X and Y, then Alice must first ensure both of those contracts are in the same shard using the yanking process, and then perform the transaction.

In comparison with S-BAC, cross-shard yanking requires the state of contracts to be transferred (temporarily) to the state of other shards, whereas S-BAC does not require object data from other shards to be stored in the state of other shards. Object data needs to be downloaded to verify transactions against their checkers, but not stored in state.

Furthermore, the unit of atomicity in Chainspace is objects rather than contracts. This means Chainspace allows for 'transparent' sharding where developers do not need to be concerned with the underlying sharding design of the system. However for Ethereum 2.0 developers to take advantage of sharding, they must craft their smart contracts in a specific way so that it spawns new 'child' smart contracts where operations can be performed in parallel. For example, a hotel management contract would spawn new child contracts to represent each hotel room, so that hotel rooms can be booked concurrently.

3.2.3 RSCoin

RSCoin [53] is a permissioned blockchain. The central bank controls all monetary supply, while mintettes (nodes authorized by the bank) manage subsets of transactions and coins. Like OmniLedger, communication between mintettes takes place indirectly, through the client—and also relies on the client to ensure completion of transactions. RSCoin has low communication overhead, and the transaction throughput scales linearly with the number of mintettes, but cannot support generic smart contracts.

This is because RSCoin assumes that objects (*e.g.,* coins) only have one owner. Thus there should not be a scenario where an honest coin owner would submit conflicting transactions to mintettes for their coins. RSCoin uses an adapted two-phase commit protocol, but without an abort phase. Therefore if a dishonest coin owner submits conflicting transactions to mintettes for their coins, their coins will effectively be locked and unspendable.

3.2.4 Elastico

Elastico [100] scales by partitioning nodes in the network into a hierarchy of committees, where each committee is responsible for managing a subset (shard) of transactions consistently through PBFT. A final committee collates sets of transactions received from committees into a final block and then broadcasts it. At the end of each epoch, nodes are reassigned to committees through proof-of-work. The block throughput scales up almost linear to the size of the network. However, Elastico cannot process multi-shard transactions.

3.2.5 RapidChain

RapidChain [157] is a sharded blockchain protocol that uses a variant of the Cuckoo rule [15] called the Bounded Cuckoo rule to assign nodes to shards. The rule states that once a new node joins the network, the set of shards are ordered by number of nodes, and the new node is assigned to one of the shards in the biggest 50% of shards. Some of the nodes in this shard are then evicted and moved to random shards in the smallest 50% of shards. This allows nodes in shards to rotate gradually without needing to reconfigure entire shards at a time, which can cause interruptions in protocol execution.

For cross-shard transactions, RapidChain uses a variant of cross-shard yanking as described in Section 3.2.2, however in RapidChain UTXOs are transferred between shards, rather than smart contracts. For example, if a user submits a transaction with two inputs and one outputs, all of the inputs are moved to the output shard before it is executed.

To achieve this on a network level, shards only maintain a connection with $\log(n)$ shards, and messages are routed to shards via the Kademlia routing algorithm [104].

3.3 System Overview

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

3.3.1 Data Model: Objects, Contracts, Transactions.

Chainspace applies aggressively the end-to-end principle [131] in relying on untrusted end-user applications to build transactions to be checked and executed. We describe below key concepts within the Chainspace data model, that developers need to grasp to use the system.

Objects are atoms that hold state in the Chainspace system. We usually refer to an object through the letter o, and a set of objects as $o \in O$. All objects have a cryptographically derived unique identifier used to unambiguously refer to the object, that we denote $\mathsf{id}(o)$. Objects also have a type, denoted as $\mathsf{type}(o)$, that determines the unique identifier of the smart contract that defines them, and a type name. In Chainspace object state is immutable. Objects may be in two meta-states, either *active* or *inactive*. Active objects are available to be operated on through smart contract procedures, while inactive ones are retained for the purposes of audit only.

Contracts are special types of objects, that contain executable information on how other objects of types defined by the contract may be manipulated. They define a set of initial objects that are created when the contract is first created within Chainspace. A contract c defines a *namespace* within which *types* (denoted as $\mathsf{types}(c)$) and a *checker* v for *procedures* (denoted as $\mathsf{proc}(c)$) are defined.

A *procedure*, p, defines the logic by which a number of objects, that may be *inputs* or *references*, are processed by some logic and *local parameters* and *local return values* (denoted as lpar and lret), to generate a number of object *outputs*. Notionally, input objects, denoted as a vector $\vec{w}$, represent state that is invalidated by the procedure; references, denoted as $\vec{r}$ represent state that is only read; and outputs are objects, or $\vec{x}$ are created by the procedure. Some of the local parameters or local returns may be secrets, and require confidentiality. We denote those as spar and sret respectively.

We denote the execution of such a procedure as:

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

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

We further require a procedure that outputs an non empty set of objects $\vec{x}$, to also take as parameters a non-empty set of input objects $\vec{w}$. Transactions that create no outputs are allowed to just take locals and references $\vec{r}$.

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

$$c.v(p, \vec{w}, \vec{r}, \mathsf{lpar}, \vec{x}, \mathsf{lret}, \mathsf{dep}) \rightarrow \{\mathsf{true}, \mathsf{false}\} \tag{3.1}$$

Note that checkers do not take any secret local parameters (spar or sret). A checker for a smart contract returns true only if there exist some secret parameters spar or sret, such that an execution of the contract procedure p, with the parameters passed to the checker alongside spar or sret, is possible as defined in Section 3.3.1. The variable dep represents the context in which the procedure is called: namely information about other procedure executions. This supports composition, as we discuss in detail in the next section.

We note that procedures, unlike checkers, do not have to be pure functions, and may be randomized, keep state or have side effects. A smart contract defines explicitly the checker $c.v$, but does not have to define procedures *per se*. The Chainspace system is oblivious to procedures, and relies merely on checkers. Yet, applications may use procedures to create valid transactions. The distinction between procedures and checkers—that do not take secrets—is key to implementing privacy-friendly contracts.

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

Once a transaction is accepted in the system it 'consumes' the input objects, that become inactive, and brings to life all new output objects that start their life

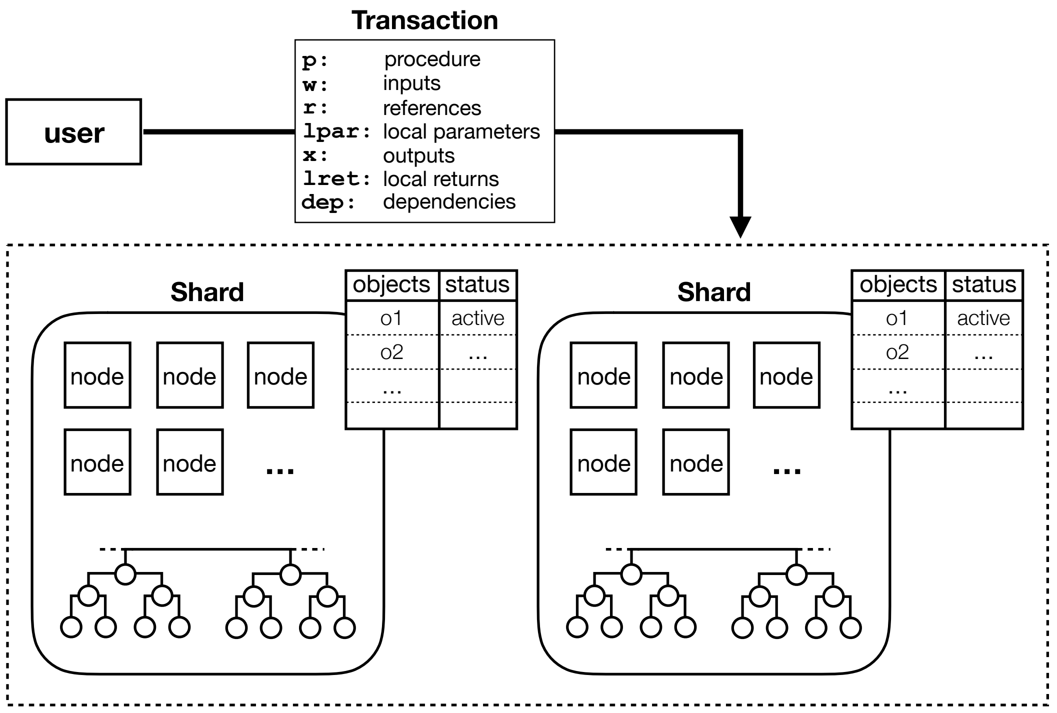


Figure 3.1: Design overview of Chainspace system, showing the interaction between users, transactions, objects and nodes in shards.

by being active. References on the other hand must be active for the transaction to succeed, and remain active once a transaction has been successfully committed.

A client packages enough information about the execution of those procedures to allow Chainspace to safely *serialize* its execution, and *atomically* commit it only if all transactions are valid according to relevant smart contract checkers.

3.3.2 System Design, Threat Model and Security Properties

We provide an overview of the system design, illustrated in Figure 6.1. Chainspace is comprised of a network of infrastructure *nodes* that manage valid objects, and ensure that only valid transactions are committed. A key design goal is to achieve scalability in terms of high transaction throughput and low latency. To this end, nodes are organized into shards that manage the state of objects, keep track of their validity, and record transactions aborted or committed. Within each shard all honest nodes ensure they consistently agree whether to accept or reject a transaction: whether an object is active or inactive at any point, and whether traces from contracts they know are valid according to their checkers. Across shards, nodes must ensure that transactions are *committed* if all shards are willing to commit the transaction, and rejected (or *aborted*) if any shards decide to abort the transaction—due to checkers returning false or objects being inactive. To satisfy these requirements, Chainspace implements S-BAC—a protocol that composes existing Byzantine agreement and atomic commit primitives in a novel way. Consensus on committing (or aborting) transactions takes place in parallel across different shards. For transparency and auditability, nodes in each shard periodically publish a signed blockchain of *checkpoints*: shards add a block (Merkle tree) of evidence including transactions processed in the current epoch, and signed promises from other nodes, to the blockchain.

Chainspace supports security properties against two distinct types of adversaries, both polynomial time bounded:

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

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

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

Given this threat model, Chainspace supports the following security properties:

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

- **Integrity.** Subject to the HS threat model, when one or more transactions are submitted only a set of valid non-conflicting transactions will be committed within the system. This includes resolving conflicts—in terms of multiple transactions using the same objects—ensuring the validity of the transactions, and also making sure that all new objects are registered as active. Ultimately, Chainspace transactions are accepted, and the set of active objects changes, as if executed sequentially—however, unlike other systems such as Ethereum [153], this is merely an abstraction and high levels of concurrency are supported.

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

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

Note that as discussed in Section 2.3.3, under a traditional blockchain threat model, the integrity property described above normally holds even in the presence of a dishonest set of consensus nodes, because full nodes do not accept blocks that contain invalid transactions. Thus ideally, integrity should be achieved even in a DS threat model. However, this is not trivial to achieve in a sharded environment, as no node is expected to fully validate the state of every shard. We address this problem is Chapter 4 with fraud and data availability proofs.

3.4 The Chainspace Application Interface

Smart contract developers in Chainspace register a smart contract c into the distributed system managing Chainspace, by defining a checker for the contract and some initial objects. Users may then submit transactions to operate on those objects in ways allowed by the checkers. Transactions represent the execution of one or more procedures from one or more smart contracts. It is necessary for all inputs to all procedures within the transaction to be active for a transaction to be executed and produce any output objects.

Transactions are *atomic*: either all their procedures run, and produce outputs, or none of them do. Transactions are also *consistent*: in case two transactions are submitted to the system using the same active object inputs, at most one of them will eventually be executed to produce outputs. Other transactions, called *conflicting*, will be aborted.

3.4.1 Representation of Transactions

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

$$
\begin{aligned}
&\text{type } \mathit{Transaction} : \mathit{Trace} \text{ list}\\
&\text{type } \mathit{Trace} : \text{Record } \{\\
&\quad c : \mathsf{id}(o), \quad p : \text{string},\\
&\quad \vec{w}, \vec{r}, \vec{x} : \mathsf{id}(o) \text{ list},\\
&\quad \mathsf{lpar}, \mathsf{lret} : \text{arbitrary data},\\
&\quad \mathsf{dep} : \mathit{Trace} \text{ list}\}
\end{aligned}
$$

To generate a set of traces composing the transaction, a *user executes on the client side all the smart contract procedures* required on the input objects, references and local parameters, and generates the output objects and local returns for every procedure—potentially also using secret parameters and returns. Thus the actual computation behind the transactions is performed by the user, and the traces forming the transaction already contain the output objects and return parameters, and sufficient information to check their validity through smart contract checkers. This design pattern is related to traditional *optimistic concurrency control* [91].

Only valid transactions are eventually committed into the Chainspace system, as specified by two validity rules *sequencing* and *checking* presented below. Transactions are considered valid within a context of a set of active objects maintained by Chainspace, denoted with α. Valid transactions lead to a new context of active objects (*e.g.,* α'). We denote this through the triplet $(\alpha, \mathit{Valid}(T), \alpha')$, which is true if the execution of transaction T is valid within the context of active objects α and generates a new context of active objects α'. The two rules are as follows:

- (Sequence rule). A '*Trace* list' (within a '*Transaction*' or list of dependencies) is valid if each of the traces are valid in sequence (*i.e.,* when executed sequentially). Further, the active objects set is updated in sequence before considering the validity of each trace.

- (Check rule). A particular '*Trace*' is valid, if the sequence of its dependencies are valid, and then in the resulting active object context, the checker for it returns `true`. A further three side conditions must hold: (1) inputs and references must be active; (2) if the trace produces any output objects it must also contain some input objects; and (3) all objects passed to the checker must be of types defined by the smart contract of this checker.

The ordering of active object sets in the validation rules result in a depth-first validation of all traces, which represents a depth-first execution and data flow dependency between them. It is also noteworthy that only the active set of objects needs to be tracked to determine the validity of new transactions, which is in the order of magnitude of active objects in the system. The much longer list of inactive objects, which grows to encompass the full history of every object in the system is not needed—which we leverage to enable better when validating transactions. It also results in a smaller amount of working memory to perform incremental audits.

A valid transaction is executed in a serialized manner, and committed or aborted atomically. If it is committed, the new set of active objects replaces the previous set; if not the set of active objects does not change. Determining whether a transaction may commit involves ensuring all the input objects are active, and all are consumed as a result of the transaction executing, as well as all new objects becoming available for processing (references however remain active). Chainspace ensures this through the distributed atomic commit protocol, S-BAC.

3.4.2 Smart Contract Composition

A contract procedure may call a transaction of another smart contract, with specific parameters and rely upon returned values. This is achieved through passing the dep variable to a smart contract checker, a validated list of traces of all the sub-calls performed. The checker can ensure that the parameters and return values are as expected, and those dependencies are checked for validity by Chainspace.

Composition of smart contracts is a key feature of a transparent and auditable computation platform. It allows the creation of a library of smart contracts that act as utilities for other higher-level contracts: for example, a simple contract can implement a cryptographic currency, and other contracts—for e-commerce for example—can use this currency as part of their logic. Furthermore, we compose smart contracts, in order to build some of the functionality of Chainspace itself as a set of ‘system’ smart contracts, including management of shards mapping to nodes, key management of shard nodes, and governance.

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

3.4.3 Reads

Besides executing transactions, Chainspace clients, need to read the state of objects, if anything, to correctly form transactions. Reads, by themselves, cannot lead to inconsistent state being accepted into the system, even if they are used as inputs or references to transactions. This is a result of the system checking the validity rules before accepting a transaction, which will reject any stale state.

Thus, any mechanism may be used to expose the state of objects to clients, including traditional relational databases, or 'no-SQL' alternatives. Additionally, any indexing mechanism may be used to allow clients to retrieve objects with specific characteristics faster. Decentralized, read-only stores have been extensively studied, so we do not address the question of reads further in this work.

3.4.4 Privacy by Design

Defining smart contract logic as checkers allows Chainspace to support privacy friendly-contracts by design. In such contracts some information in objects is not in the clear, but instead either encrypted using a public key, or committed using a secure commitment scheme as [119]. The transaction only contains a valid proof that the logic or invariants of the smart contract procedure were applied correctly or hold respectively, and can take the form of a zero-knowledge proof, or a Succinct Argument of Knowledge (SNARK). Then, generalizing the approach of [107], the checker runs the verifier part of the proof or SNARK that validates the invariants of the transactions, without revealing the secrets within the objects to the verifiers.

3.5 The Chainspace System Design

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

3.5.1 High-Integrity Data Structures

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

Hash-DAG Structure

Objects and transactions naturally form a directed acyclic graph (DAG): given an initial state of active objects a number of transactions render their inputs invalid, and create a new set of outputs as active objects. These may be represented as a directed graph between objects, transactions and new objects and so on. Each object may only be created by a single transaction trace, thus cycles between future transactions and previous objects never occur. We prove that output object identifiers resulting from valid transactions are fresh (see Theorem 1). Hence, the graph of objects inputs, transactions and objects outputs form a DAG, that may be indexed by their identifiers.

We leverage this DAG structure, and augment it to provide a high-integrity data structure. Our principal aim is to ensure that given an object, and its identifier, it is possible to unambiguously and unequivocally check all transactions and previous (now inactive) objects and references that contribute to the existence of the object. To achieve this we define as an identifier for all objects and transactions a cryptographic hash that directly or indirectly depends on the identifiers of all state that contributed to the creation of the object.

Specifically, we define a function $\mathsf{id}(\mathit{Trace})$ as the identifier of a trace contained in transaction T. The identifier of a trace is a cryptographic hash function over the name of contract and the procedure producing the trace; as well as serialization of the input object identifiers, the reference object identifiers, and all local state of the transaction (but not the secret state of the procedures); the identifiers of the trace's dependencies are also included. Thus all information contributing to defining the Trace is included in the identifier, except the output object identifiers.

We also define the $\mathsf{id}(o)$ as the identifier of an object o. We derive this identifier through the application of a cryptographic hash function, to the identifier of the trace that created the object o, as well as a unique name assigned by the procedures creating the trace, to this output object. (Unique in the context of the outputs of this procedure call, not globally, such as a local counter.)

An object identifier $\mathsf{id}(o)$ is a high-integrity handle that may be used to authenticate the full history that led to the existence of the object o. Due to the collision resistance properties of secure cryptographic hash functions an adversary is not able to forge a past set of objects or transactions that leads to an object with the same identifier. Thus, given $\mathsf{id}(o)$ anyone can verify the authenticity of a trace that led to the existence of o.

A very important property of object identifiers is that future transactions cannot re-create an object that has already become inactive. Thus checking object validity only requires maintaining a list of active objects, and not a list of past inactive objects:

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

Proof. We argue this property by induction on the serialized application of valid transactions, and for each transaction by structural induction on the two validity rules. Assuming a history of $n-1$ transactions for which this property holds we consider transaction n. Within transaction n we sequence all traces and their dependencies, and follow the data flow of the creation of new objects by the 'check' rule. For two objects to have the same $\mathsf{id}(o)$ there need to be two invocations of the check rule with the same contract, procedure, inputs and references. However, this leads to a contradiction: once the first trace is checked and considered valid the active input objects are removed from the active set, and the second invocation becomes invalid. Thus, as long as object creation procedures have at least one input (which is ensured by the side condition) the theorem holds, unless an adversary can produce a hash collision. The inductive base case involves assuming that no initial objects start with the same identifier – which we can ensure axiomatically. □

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

Shard Blockchains

Each node in Chainspace, that is entrusted with preserving integrity, associates with its shard a blockchain. Periodically, peers within a shard consistently agree to seal a *checkpoint*, as a block of transactions into their blockchains. They each form a Merkle tree containing all transactions that have been accepted or rejected in sequence by the shard since the last checkpoint was sealed. Then, they extend their blockchain by hashing the root of this Merkle tree and a block sequence number, with the head hash of the chain so far, to create the new head of the blockchain. Each peer signs the new head of their chain, and shares it with all other peers in the shard, and anyone who requests it. For strong auditability additional information, besides committed or aborted transactions, has to be included in the Merkle tree: node should log any promise to either commit or abort a transaction from any other peer in any shard (the prepared(T,*) statements explained in the next sections).

All honest nodes within a shard independently create the same chain for a checkpoint, and a signature on it—as long as the consensus protocols within the shards are correct. We say that a checkpoint represents the decision of a shard, for a specific sequence number, if at least $f+1$ signatures of shard nodes sign it. On the basis of these blockchains we define a *light audit* and a *full audit* of the Chainspace system. Light audits can be performed by light clients, where full audits can be performed by full nodes, as described in Section 2.3.3.

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

A *full audit* involves replaying all transactions processed by the shard, and ensuring that (1) all transactions were valid according to the checkers the shard executed; (2) the objects input or references of all committed transactions were all active (see rules in Section 3.4.1); and (3) the evidence received from other shards supports committing or aborting the transactions. To do so an auditor downloads the full blockchain representing the decisions of the shard from the beginning of time, and re-executes all the transactions in sequence. This is possible, since—besides their secret signing keys—peers in shards have no secrets, and their execution is deterministic once the sequence of transactions is defined. Thus, an auditor can re-execute all transactions in sequence, and check that their decision to commit or abort them is consistent with the decision of the shard. Doing this, requires any inter-shard communication (namely the promises from other shards to commit or abort transactions) to be logged in the blockchain, and used by the auditor to guide the re-execution of the transactions. A full audit needs to re-execute all transactions and requires evidence of size $O(N)$ in the number N of transactions. This is costly, but may be done incrementally as new blocks of shard decisions are created.

3.5.2 Distributed Architecture & Consensus

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

Chainspace uses sharding strategies to ensure scalability: a public function $\phi(o)$ maps each object o to a set of nodes, we call a *shard*. These nodes collectively are entrusted to manage the state of the object, keep track of its validity, record transactions that involve the object, and eventually commit at most one transaction consuming the object as input and rendering it inactive. However, nodes must only record such a transaction as committed if they have certainty that all other nodes have, or will in the future, record the same transaction as consuming the object. We call this distributed algorithm the *consensus* algorithm within the shard.

For a transaction T we define a set of *concerned nodes*, $\Phi(T)$ for a transaction structure T. We first denote as ζ the set of all objects identifiers that are input into or referenced by any trace contained in T. We also denote as ξ the set of all objects that are output by any trace in T. The function $\Phi(T)$ represents the set of nodes that are managing objects that should exist, and be active, in the system for T to succeed. More mathematically, $\Phi(T) = \bigcup\{\phi(o_i) | o_i \in \zeta \setminus \xi\}$, where $\zeta \setminus \xi$ represents the set of objects input but not output by the transaction itself (its free variables). The set of concerned peers thus includes all shard nodes managing objects that already exist in Chainspace that the transaction uses as references or inputs.

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

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

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

Theorem 2 ensures that the set of concerned nodes, includes nodes that manage objects from all contracts represented in a transaction. Chainspace leverages this to distribute the process of rule validation across peers in two ways:

- For any existing object o in the system, used as a reference or input within a transaction T, only the shard nodes managing it, namely in $\phi(o)$, need to check that it is active (as part of the 'check' rule in Section 3.4.1).

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

However, all shards containing concerned nodes for T need to ensure that all others have performed the necessary checks before committing the transaction, and creating new objects.

To ensure that concerned nodes in each shards do not reach an inconsistent state for the accepted transactions, we design an open, scalable and decentralized mechanism to perform *Sharded Byzantine Atomic Commit* or S-BAC.

3.5.3 Sharded Byzantine Atomic Commit (S-BAC)

Chainspace implements the previously described intra-shard consensus algorithm for transaction processing in the *Byzantine* and *asynchronous* setting, through the *Sharded Byzantine Atomic Commit* (S-BAC) protocol, that combines two primitive protocols: *Byzantine Agreement* and *atomic commit*.

- *Byzantine agreement* ensures that all honest members of a shard of size $3f+1$, agree on a specific common sequence of actions, despite some f malicious nodes within the shard. It also guarantees that when agreement is sought, a decision or sequence will eventually be agreed upon. The agreement protocol is executed within each shard to coordinate all nodes. We use MOD-SMART [141] implementation of PBFT for state machine replication that provides an optimal number of communications steps (similar to PBFT [43]). This is achieved by replacing reliable broadcast with a special leader-driven Byzantine consensus primitive called Validated and Provable Consensus (VP-Consensus).

- *Atomic commit* is run across all shards managing objects relied upon by a transaction. It ensures that each shard needs to accept to commit a transaction, for the transaction to be committed; even if a single shard rejects the

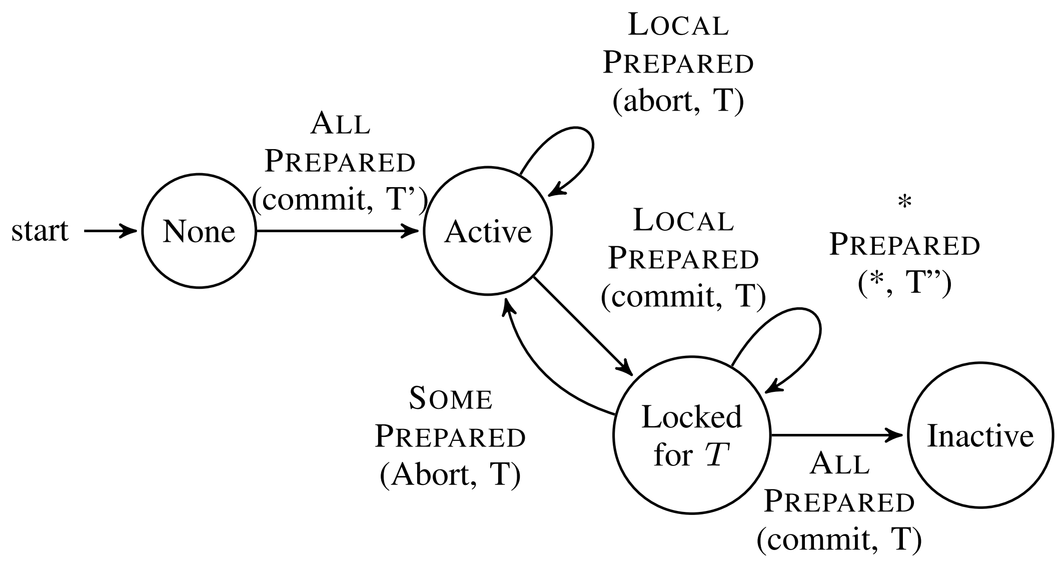


Figure 3.2: The state machine representing the active, locked and inactive states for any object within Chainspace. Each node in a shard replicates the state of the object, and participates in a consensus protocol that allows it to derive the invariants "Local prepared", "All prepared", and "Some prepared" to update the state of an object.

transaction, then all agree it is rejected. We propose the use of a simple two-phase commit protocol [26], composed with an agreement protocol to achieve this—loosely inspired by Lamport and Gray [72]. This protocol was the first to reconcile the needs for distributed commit, and replicated consensus (but only in the non-Byzantine setting).

S-BAC composes the above primitives in a novel way to ensure that shards process safely and consistently all transactions. Figure 3.3 illustrates a simple example of the S-BAC protocol to commit a single transaction with two inputs and one output that we may use as an example. The corresponding object state transitions have been illustrated in Figure 3.2. The combined protocol has been described below. For ease of understanding, in our description we state that all messages are sent and processed by shards. In reality, some of these are handled by a designated node in each shard—the BFT-Initiator—as we discuss at the end of this section.

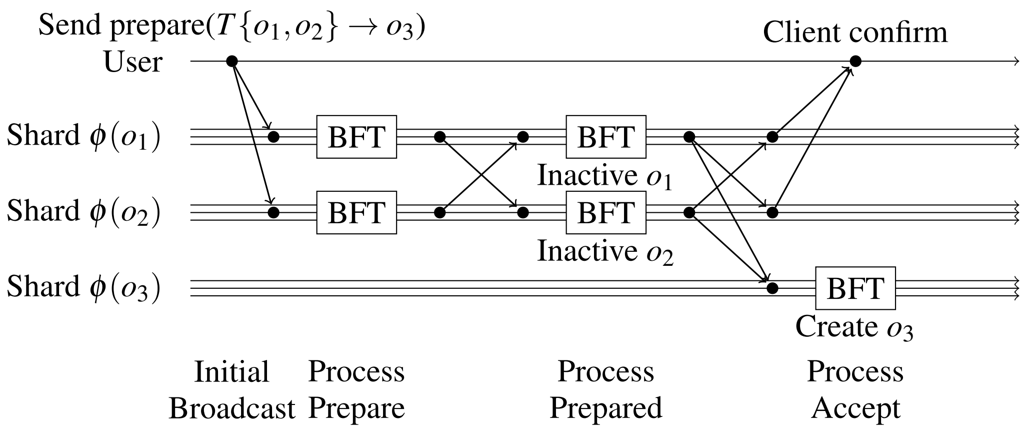


Figure 3.3: S-BAC for a transaction T with two inputs (o_1, o_2) and one output object (o_3). The user sends the transaction to all nodes in shards managing o_1 and o_2. The BFT-Initiator takes the lead in sequencing T, and emits 'prepared(accept, T)' or 'prepared(abort, T)' to all nodes within the shard. Next the BFT-Initiator of each shard assesses whether overall 'All proposed(accept, T)' or 'Some proposed(abort, T)' holds across shards, sequences the accept(T,*), and sends the decision to the user. All cross-shard arrows represent a multicast of all nodes in one shard to all nodes in another.

Initial Broadcast (Prepare)

A user acts as a transaction initiator, and sends 'prepare(T)' to at least one honest concerned node for transaction T. To ensure at least one honest node receives it, the user may send the message to $f+1$ nodes of a single shard, or $f+1$ nodes in each concerned shard.

Sequence Prepare

Upon a message 'prepare(T)' being received, nodes in each shard interpret it as the initiation of a two-phase commit protocol performed across the concerned shards. The shard locally sequences 'prepare(T)' message through the Byzantine consensus protocol.

Process Prepare

Upon the first action 'prepare(T)' being sequenced through BFT consensus in a shard, nodes of the shard implicitly decide whether it should be committed or aborted. Since all honest nodes in the shard have a consistent replica of the full sequence of actions, they will all decide the same consistent action following 'prepare(T)'.

Transaction T is to be committed if it is valid according to the usual rules (see Section 3.4.1), in brief: (1) the objects input or referenced by T in the shard are active, (2) there is no other instance of the two-phase commit protocol on-going concerning any of those objects (no locks held) and (3) if T is valid according to the validity rules, and the smart contract checkers in the shard. Only the checkers for types of objects held by the shard are checked by the shard.

If the decision is to commit, the shard broadcasts to all concerned nodes 'prepared(T,commit)', otherwise it broadcasts 'prepared(T, abort)'—along with sufficient signatures to convince any party of the collective shard decision (we denote this action as LOCALPREPARED(*, T)). The objects used or referenced by T are 'locked' (Figure 3.2) in case of a 'prepared commit' until an 'accept' decision on the transaction is reached, and subsequent transactions concerning them will be aborted by the shard. Any subsequent 'prepare(T'')' actions in the sequence are ignored, until a matching accept(T, abort) is reached to release locks, or forever if the transaction is committed.

Process Prepared (Accept or Abort)

Depending on the decision of 'prepare(T)', the shard sequences 'accept(T,commit)' or 'accept(T,abort)' through the atomic commit protocol across all the concerned shards—along with all messages and signatures of the bundle of 'prepared' messages relating to T proving to other shards that the decision should be 'accept(T,commit)' or 'accept(T,abort)' according to its local consensus. If it receives even a single 'LOCALPREPARED(T,abort)' from another shard it instead will move to reach consensus on 'accept(T, abort)' (denoted as SOMEPREPARED(abort,T)). Otherwise, if all the shards respond with 'LOCALPREPARED(T,commit)' it will reach a consensus on ALLPREPARED(commit,T). The final decision is sent to the user, along with all messages and signatures of the bundle of 'accept' messages relating to T proving that the final decision should be to commit or abort according to responses from all concerned shards.

It is possible, that a shard hears a prepared message for T before a prepare message, due to unreliability, asynchrony or a malicious user. In that case the shard assumes that a 'prepare(T)' message is implicit, and sequences it.

Process Accept

When a shard sequences an 'accept(T, commit)' decision, it sets all objects that are inputs to the transaction T as being inactive (Figure 3.2). It also creates any output objects from T via BFT consensus that are to be managed by the shard. If the output objects are not managed by the shard, the shard sends requests to the concerned shards to create the objects. On the other hand if the shard decision is 'accept(T, abort)', all nodes release locks held on inputs or references of transaction T. Thus those objects remain active and may be used by other transactions.

As previously mentioned, some of the messages in S-BAC are handled by a designated node in each shard called the BFT-Initiator. Specifically, the BFT-Initiator drives the composed S-BAC protocol by sending 'prepare(T)' and then 'accept(T, *)' messages to reach BFT consensus within and across shards. It is also responsible for broadcasting consensus decisions to relevant parties. The protocol supports a two-phase process to recover from a malicious BFT-Initiator that suppresses transactions. As nodes in a shard hear all messages, they wait for the BFT-Initiator to act on it until they time out. They first send a reminder to the BFT-Initiator along with the original message to account for network losses. Next they proceed to wait; if they time out again, other nodes perform the action of BFT-Initiator which is idempotent.

Out of scope in this chapter is the prevention of replay attacks within S-BAC. Specifically, an adversary may capture and replay 'prepared(T,commit)' or 'accept(T,commit)' messages from previous rounds of S-BAC for T, to cause shards to accept transactions that do not have consensus, or to cause output shards to create the same object multiple times. In a separate work co-authored by the author of this thesis, [138], we discuss how to handle message replays in cross-shard transaction protocols including S-BAC and OmniLedger's Atomix protocol.

3.5.4 Concurrency & Scalability

Each transaction T involves a fixed number of *concerned nodes* $\Phi(T)$ within Chainspace, corresponding to the shards managing its inputs and references. If two transactions T_0 and T_1 have disjoint sets of concerned nodes ($\Phi(T_0) \cap \Phi(T_1) = \emptyset$) they cannot conflict, and are executed in parallel or in any arbitrary order. If however, two transactions have common input objects, only one of them is accepted by all nodes. This is achieved through the S-BAC protocol. It is local, in that it concerns only nodes managing the conflicting transactions, and does not require a global consensus.

From the point of view of scalability, Chainspace capacity grows linearly as more shards are added, subject to transactions having on average a constant, or sub-linear, number of inputs and references (see Figure 3.5). Furthermore, those inputs must be managed by different nodes within the system to ensure that load of accepting transactions is distributed across them.

3.5.5 System Contracts

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

Shard Management

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

Smart Contract Management

Chainspace is also 'open' in the sense that anyone may create a new smart contract, and this process is implemented using the ManageContracts smart contract. ManageContracts implements three types: MC.Token, MC.Mapping and MC.Contract. It also implements at least one procedure, MC.create that takes a binary representing a checker for the contract, an initialization procedure name that creates initial objects for the contract, and the singleton token object. It then creates a number of outputs: one object of type MC.Token for use to create further contracts; an object of type MC.Contract representing the contract, and containing the checker code, and a mapping object MC.mapping encoding the mapping between objects of the contract and shards within the system. Furthermore, the procedure MC.create calls the initialization function of the contract, with the contract itself as reference, and the singleton token, and creates the initial objects for the contract.

Note that this simple implementation for ManageContracts does not allow for updating contracts. The semantics of such an update are delicate, particularly in relation to governance and backwards compatibility with existing objects. We leave the definitions of more complex, but correct, contracts for managing contracts as future work. In our first implementation we have hardcoded ManageShards and ManageContracts.

3.6 Security and Correctness

3.6.1 Security & Correctness of S-BAC

The S-BAC protocol guarantees a number of key properties, on which rest the security of Chainspace, namely *liveness*, *consistency*, and *validity*. Before proceeding with stating those properties in details, and proving them we note three key invariants, that nodes may decide:

- LOCALPREPARED(commit / abort, T): A node considers that LOCALPREPARED(commit / abort, T) for a shard holds, if it receives at least $f+1$ distinct signed messages from nodes in the shard, stating 'prepared(commit, T)' or 'prepared(abort, T)' respectively. As a special case a node automatically concludes LOCALPREPARED(commit / abort, T) for a shard it is a member of, if all the preconditions necessary to provide that answer are present when an 'prepare(T)' is sequenced.

- ALLPREPARED(commit, T): A node considers that 'ALLPREPARED(commit, T)' holds if it believes that 'LOCALPREPARED(commit, T)' holds for all shards with concerned nodes for T. Note this may only be decided after reaching a conclusion (e.g. through receiving signed messages) about all shards.

- SOMEPREPARED(abort, T): A node considers that 'SOMEPREPARED(abort, T)' holds if it believes that 'LOCALPREPARED(abort, T)' holds for at least one shard with concerned nodes for T. This may be concluded after only reaching a conclusion for a single shard, including the shard the node may be part of.

Liveness ensures that transactions make progress once proposed by a user, and no locks are held indefinitely on objects, preventing other transactions from making progress.

Theorem 3. *Liveness: Under the 'honest shards' threat model, a transaction T that is proposed to at least one honest concerned node, will eventually result in either being committed or aborted, namely all parties deciding accept(commit, T) or accept(abort, T).*

Proof. We rely on the standard liveness properties of the underlying Byzantine agreement: *i.e.,* shards with only up to f faulty nodes will reach a consensus on a sequence eventually. We also rely on the broadcast from nodes of shards to all other nodes of shards, including the shards that manage transaction outputs. Assuming prepare(T) has been given to an honest node, it will be sequenced within an honest shard BFT sequence, and thus a prepared(commit, T) or prepared(abort, T) will be sent from the $2f+1$ honest nodes of this shard, to the $2f+1$ nodes of the other concerned shards. Upon receiving these messages the honest nodes from other shards will schedule a prepare(T) message within their shards, and the BFT will eventually sequence it. Thus the user and all other honest concerned nodes will receive enough 'prepared' messages to decide whether to proceed with 'ALLPREPARED(commit, T)' or 'SOMEPREPARED(abort, T)' and proceed with sequencing them through BFT. Eventually, each shard will sequence those, and decide on the appropriate 'accept'. □

The second key property ensures that the execution of valid transactions could be serialized, and thus is correct.

Theorem 4. *Consistency: Under the 'honest shards' threat model, no two conflicting transactions, namely transactions sharing the same input will be committed. Furthermore, a sequential execution for all transactions exists.*

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

Theorem 5. *Validity: Under the 'honest shards' threat model, a transaction may only be committed if it is valid according to the smart contract checkers matching the traces of the procedures it executes.*

Proof. A Chainspace transaction is committed only if some nodes conclude that 'ALLPREPARED(commit, T)', which presupposes all shards have provided enough evidence to conclude 'LOCALPREPARED(commit, T)' for each of them. The concerned nodes include at least one shard per input or reference object for the transaction; for any contract c represented in the transaction, at least one of those shards will be managing object from that contract. Each shard checks the validity rules for the objects they manage (ensuring they are active, and not locked) and the contracts those objects are part of (ensuring the calls to c pass its checker) in order to LOCALPREPARED(accept, T). Thus if all shards say LOCALPREPARED(accept, T) to conclude that 'ALLPREPARED(commit, T)', all object have been checked as active, and all the contract calls within the transaction have been checked by at least one shard—whose decision is honest due to at most f faulty nodes. If even a single object is inactive or locked, or a single trace for a contract fails to check, then the honest nodes in the shard will emit 'prepared(abort, T)' upon sequencing 'prepare(T)', and the final decision will be 'SOMEPREPARED(abort, T)'. □

3.6.2 Auditability

In the previous sections we show that if each shard contains at most f faulty nodes (honest shard model), the S-BAC protocol guarantees consistency and validity. In this section we argue that if this assumption is violated, i.e. one or more shards contain more than f Byzantine nodes each, then honest shards can detect faulty shards. Namely, enough auditing information is maintained by honest nodes in Chainspace to detect inconsistencies and attribute them to specific shards (or nodes within them).

The rules for transaction validity are summarized in Section 3.4.1. Those rules are checked in a distributed manner: each shard keeps and checks the active or inactive state of objects assigned to it; and also only the contract checkers corresponding to the type of those objects. An honest shard emits a proposed(T, commit) for a transaction T only if those checks pass, and proposed(T, abort) otherwise or if there is a lock on a relevant object. A dishonest shard may emit proposed(T, *) messages arbitrarily without checking the validity rules. By definition, an invalid transaction is one that does not pass one or more of the checks defined in Section 3.4.1 at a shared, for which the shard has erroneously emitted a proposed(T, commit) message.

Theorem 6. *Auditability: A malicious shard (with more than f faulty nodes) that attempts to introduce an invalid transaction or object into the state of one or more honest shards, can be detected by an auditor performing a full audit of the Chainspace system.*

Proof. We consider two blockchains from two distinct shards. We define the pair of them as being valid if (1) they are each valid under full audit, meaning that a re-execution of all their transactions under the messages received yields the same decisions to commit or abort all transactions; and (2) if all prepared(T,*) messages in one chain are compatible with all messages seen in the other chain. In this context 'compatible' means that all prepared(T,*) statements received in one shard from the other represent the 'correct' decision to commit or abort the transaction T in the other shard. An example of incompatible message would result in observing a proposed(T, commit) message being emitted from the first shard to the second, when in fact the first shard should have aborted the transaction, due to the checker showing it is invalid or an input being inactive.

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

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

3.7 Implementation & Evaluation

We implemented a prototype of Chainspace in $\sim$10,000 lines of Python and Java code. The implementation consists of two components: a Python contracts environment and a Java node. We have released the code as an open-source project.[2]

3.7.1 Python Contract Environment

The Python contracts environment allows developers to write, deploy and test smart contracts. These are deployed on each node by running the Python script for the contract, which starts a local web service for the contract's checker. The contract's checker is then called though the web service. The environment provides a framework to allow developers to write smart contracts without knowledge of the underlying implementation, and provides an auto-generated checker for simple contracts.

3.7.2 Java Node Implementation

The Java node implements a shard replica that accepts incoming transactions from clients and initiates, and executes, the S-BAC protocol. For BFT consensus, we use

[1]The first reported use is 1340AD [95].

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

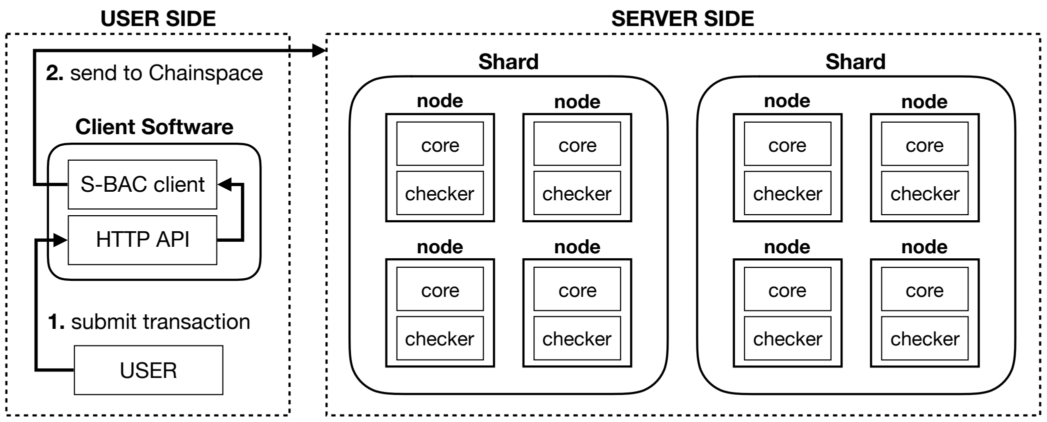


Figure 3.4: Diagram illustrating the implementation of a Chainspace system with two shards managing four nodes each. The user submits the transaction to its local S-BAC client through a built-in HTTP API (arrow 1). Then, this S-BAC client sends the transaction to Chainspace (arrow 2).

the BFT-SMARt [27] Java library—one of the very few maintained open source libraries for BFT.

To communicate with Chainspace, end users also run an S-BAC–enabled client. First, she creates a transaction through the Python environments using a smart contract. She then submits the transaction to its S-BAC client through the HTTP API as indicated in Figure 3.4, that sends the transaction to Chainspace according to the BFT-SMARt protocol.

A node is composed of a server divided in two parts: the core and the checker. To communicate with other nodes, each node also contains an S-BAC client. When a transaction is received, the core is in charge of verifying that the input objects and references are active (neither locked nor inactive). Then, the node runs the checker associated with the contract, in an isolated container. (The checker is provided by the contract's creator when the node starts up, and interfaces with the node through an HTTP API.) When the client submits a transaction with dependencies, the core recursively checks each dependent transaction first, and the top-level transaction at last (similar to depth-first search algorithm).

3.7.3 Performance Measurements

We evaluated the performances and scalability of our implementation of Chainspace, through deployments on Amazon EC2 containers. We launched up to 96 nodes on *t2.medium* virtual machines, each containing 8 GB of RAM on 2 virtual CPUs and running GNU/Linux Debian 8.1. We sent transactions to the network from a Chainspace client running on a t2.xlarge virtual machine, containing 16 GB of RAM and 4 virtual CPUs, also running GNU/Linux Debian 8.1. In our tests, we map objects to shards randomly using the mapping function $\phi(o) = id(o) \bmod K$ where K is a constant representing the number of shards and $id(o)$ is the SHA256 hash of the object.

We first measure the effect of the number of shards on transaction throughput (Figure 3.5). The transaction throughput of Chainspace scales linearly with the number of shards: with 4 nodes per shard, the number of transactions per second (t/s) increases on average by 22 for 1-input transactions for each shard added. This is because as inputs are randomly assigned to shards based on their hashes, the transaction processing load is spread out over a larger number of shards.

Next we investigate the effect of shard size (the number of nodes per shard) on transaction throughput (Figure 3.8). We fix the number of shards to 2, and increase the number of nodes per shard from 2 to 48. With BFT-SMART configured for $3f+1$ fault tolerance, we observe an expected gracious decrease in transaction throughput: for each node added, the throughput reduces on average by 1.6 transactions per second. This is because in order for a BFT-SMART node to realise consensus for a message, it must receive a result from at least $f+1$ nodes. Thus, the bottleneck is the latency of the $f+1$th node with the highest response time.

Another factor that can potentially affect transaction throughput is the number of inputs per transaction: the more shards touched by the transaction inputs, the longer it will take to run S-BAC among all the concerned shards. In Figure 3.6, we study how the number of inputs per transaction affects transaction throughput. We measure this for 5 shards, varying the number of inputs per transaction from 1 to 10, and the inputs are randomly mapped to shards as previously stated. The transaction throughput decreases asymptotically until it becomes stable at around 40 transactions per second. This is because S-BAC's maximum time in processing transactions is capped at the time it takes to process transactions that touch all the 5 shards. Increasing the number of inputs does not further deteriorate the transaction throughput.

Finally, we measure the client-perceived latency—the time from when a client submits a transaction until it receives a decision about whether the transaction has been committed—under varying system loads expressed in terms of transactions received per second. Figure 3.7 shows the effect of transactions received by the system per second (all 1-input transactions) on client-perceived latency for 2 shards, each having 4 nodes. Recall from Figure 3.5 that the average throughput for a Chainspace system with similar configuration is 75 1-input transactions per second. Consequently, we observe in Figure 3.8 that the increase in latency with varying system loads is smaller for 20 t/s–60 t/s (average 69 ms), but the values start to get bigger after 60 t/s (average 210 ms). This is when the system reaches its maximum transaction throughput, causing a backlog of transactions to be processed.

3.8 Conclusions

We presented the design of Chainspace—an open, distributed ledger platform for high-integrity and transparent processing of transactions. However, unlike existing smart-contract based systems such as Ethereum [153], it offers high scalability through sharding across nodes using a novel distributed atomic commit protocol called S-BAC, while offering high auditability. We presented implementation and evaluation of S-BAC on a real cloud-based testbed under varying transaction loads and showed that Chainspace's transaction throughput scales linearly with the number of shards by up to 22 transactions per second for each shard added, handling up to 350 transactions per second with 15 shards. As such it offers a competitive alternative to both centralized and permissioned systems, as well as fully peer-to-peer, but unscalable systems like Ethereum.

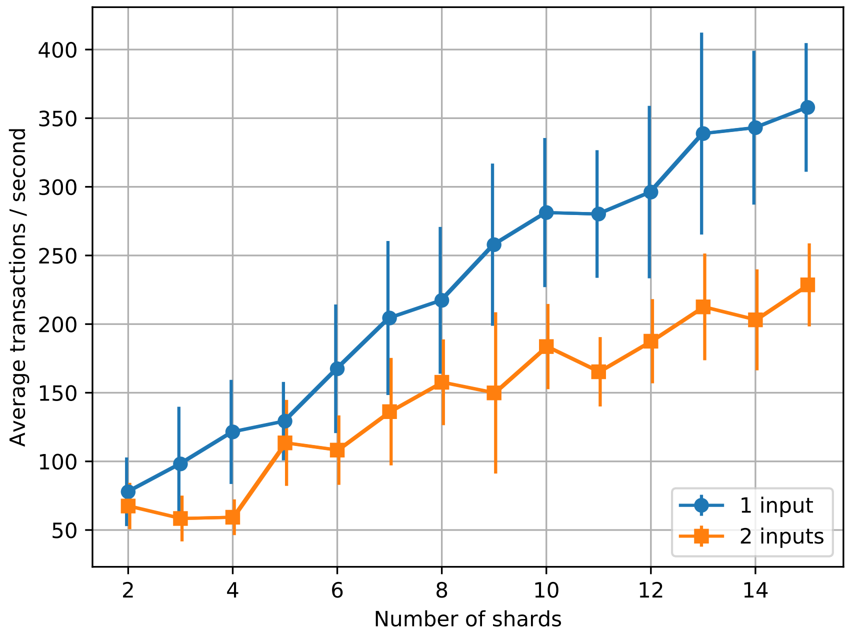


Figure 3.5: The effect of the number of shards on transaction throughput. (Nodes per shard: 4, input-to-shard mapping: random. Repeats: 20. Error bars show standard deviation.)

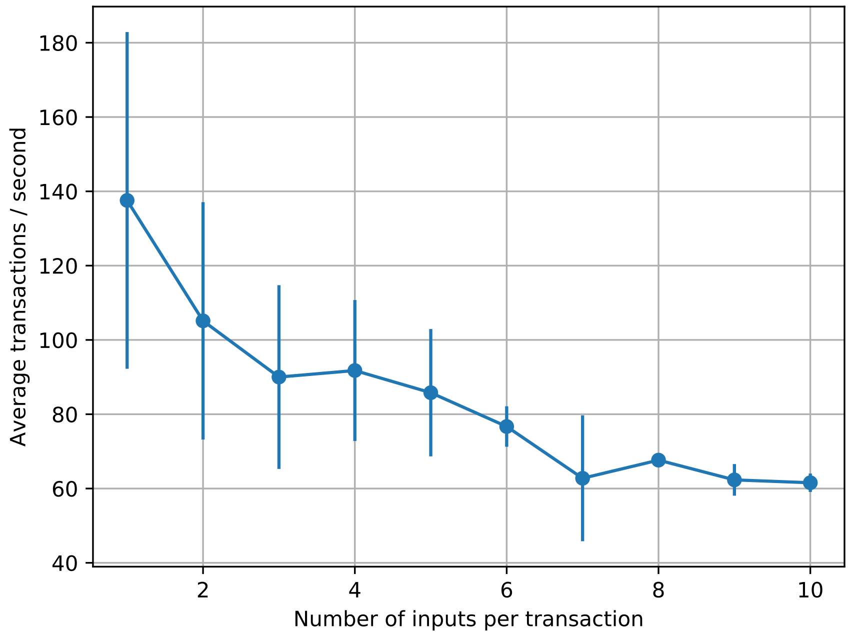


Figure 3.6: The effect of the number of inputs per transaction on transaction throughput. (Shards: 2, nodes per shard: 4, input-to-shard mapping: random. Repeats: 20.)

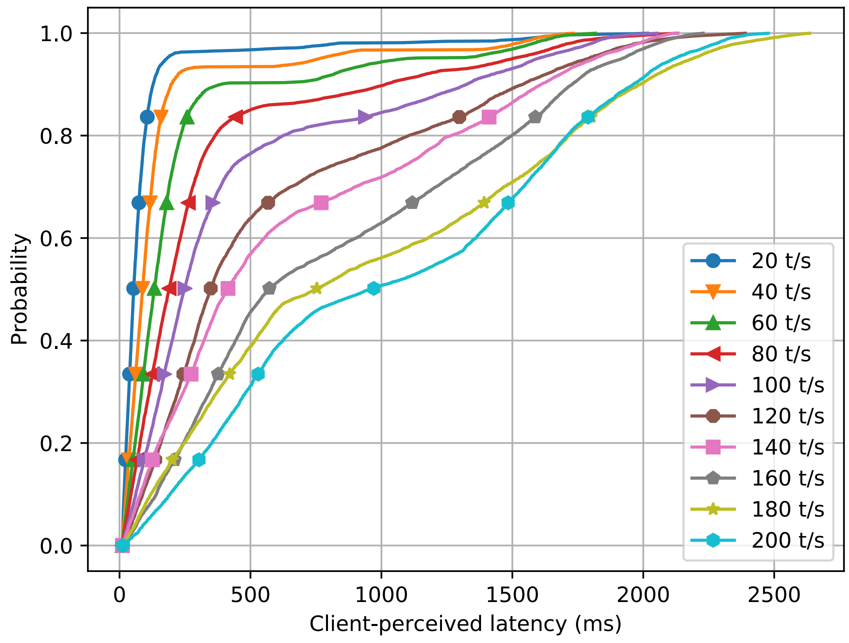


Figure 3.7: The cumulative distribution function of delay for the client to receive a final commit or abort response, for varying system load. (Shards: 5, nodes per shard: 4, inputs per transaction: 1, input-to-shard mapping: random. Repeats: 5. Error bars show standard deviation.)

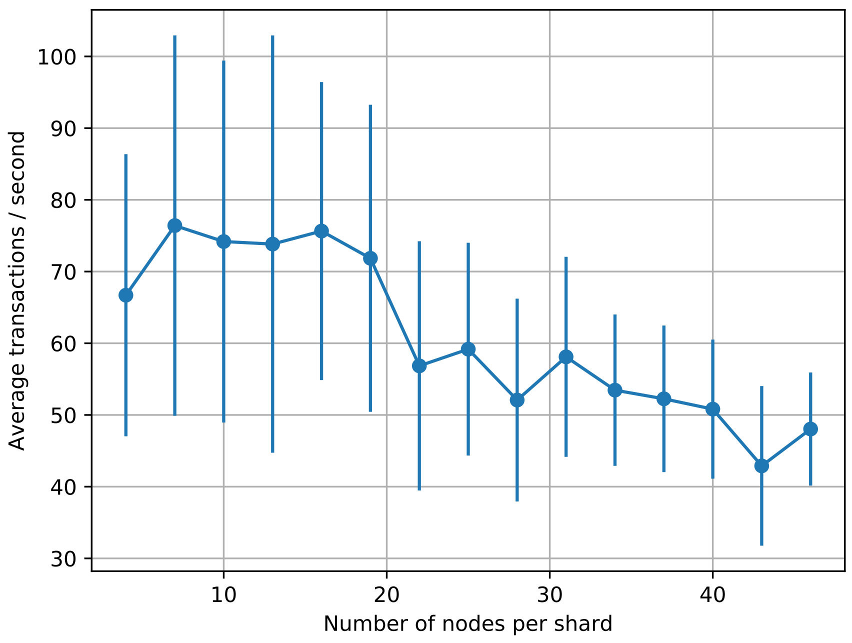


Figure 3.8: The effect of the number of nodes per shard on transaction throughput. (Shards: 2, inputs per transaction: 1, input-to-shard mapping: random. Repeats: 20. Error bars show standard deviation.)

Chapter 4

Fraud and Data Availability Proofs: Maximising Light Client Security and Scaling Blockchains with Dishonest Majorities

I'm not upset that you lied to me, I'm upset that from now on I can't believe you.

Friedrich Nietzsche

4.1 Introduction and Motivation

While increasing on-chain capacity–whether it be via sharding or simply increasing the block size–would yield higher transaction throughput, there are concerns that this creates a trade-off that would decrease decentralisation and security, because it would increase the resources required to fully download and validate the blockchain. Thus fewer users would be able to afford to run full nodes that independently validate the blockchain, requiring users to instead run light clients that assume that the chain favoured by the blockchain's consensus algorithm abides by the protocol rules for transaction validity [102]. Recall in Section 2.3.3 that light clients operate well under normal circumstances, but have weaker assurances when the majority of the consensus (*e.g.,* miners or block producers) is dishonest (also known as a '51% attack', or also referred to as the 'Dishonest Shards' threat model in Section 3.3.2). For example, whereas a dishonest majority in the Bitcoin or Ethereum network can at present only censor, reverse or reorder transactions, if all clients are using light nodes, a majority of the consensus would be able to collude together to generate blocks that contain contain invalid transactions that, for example, create money out of thin air, and light nodes would not be able to detect this. On the other hand, full nodes would reject those invalid blocks immediately.

In this chapter, we decrease the on-chain capacity vs. security trade-off by making it possible for light clients to receive and verify fraud proofs of invalid blocks from full nodes, so that they too can reject them, assuming that there is at least one honest full node willing to generate fraud proofs to be propagated within a maximum network delay. We also design a data availability proof system, a necessary complement to fraud proofs, so that light clients have assurance that the block data required for full nodes to generate fraud proofs from is available, given that there is a minimum number of honest light clients to reconstruct missing data from blocks. We implement and evaluate the security and efficiency of our overall design.

Our work also plays a key role in efforts to scale blockchains with sharding (Chapter 3), as in a sharded system no single node in the network is expected to download and validate the state of all shards, and thus fraud proofs are necessary to detect invalid blocks from malicious shards (*i.e.,* the violation of the integrity property by dishonest shards as described in Section 3.3.2).

4.2 Related Work

4.2.1 Alerts and Fraud Proofs

The original Bitcoin whitepaper [109] briefly mentions the possibility of 'alerts', which are messages sent by full nodes to alert light clients that a block is invalid, prompting them to download the full block to verify the inconsistency. This suffers from a denial-of-service problem, as false alerts could cause light clients to download and verify every block in the chain, which would require the same resources as a full node.

There have been online discussions about how one may go about designing 'compact fraud proof' systems [126, 145], where only one specific invalid transaction in a block needs to be sent to prove that a block is invalid. However no complete design that deals with all transaction invalidity cases and data availability has been proposed. These earlier systems have taken the approach of attempting to design a fraud proof for each possible way to create a block that violates the transaction validity rules (*e.g.,* double spending inputs, mining a block with a reward too high, etc), whereas this chapter generalises the blockchain into a state transition system with only one fraud proof.

To prove invalid state transitions, we use similar techniques to TrueBit [144] and Arbitrum [80], which are layer two protocols where smart contracts can be executed off-chain by the parties involved. The execution of smart contracts are divided into many state transitions. In the event that a malicious party has incorrectly executed the result of a smart contract execution, a dispute resolution protocol can be triggered by another party to pin-point the specific state transition where their execution diverged. The other party can provide a compact proof of what the correct result of the state transition should be (known as a 'one-step proof' in Arbitrum). This proof is then executed by an on-chain smart contract to resolve the dispute. The proof consists of a state commitment (*i.e.,* state root) representing the starting state of the execution, and the execution instruction. The instruction is then re-executed to determine what the resulting post-execution state commitment should be.

4.2.2 Data Availability

One of the key unsolved challenges of designing a fraud proof system is the data availability problem. If a block producer releases a block header that contains a Merkle root of transactions, but does not publish the transaction data itself, full nodes would not have the data to generate a fraud proof. Therefore a block producer could prevent fraud proofs from being generated. To prevent this, there ought to be a way for light clients to be able to verify all the transaction data in a block is available to the network, without downloading all the data itself.

To solve this, we utilise erasure coding, which allows data to be encoded in such a way that the entire data can be recovered from only a subset of the data. By requiring block producers to commit to the erasure coded version of block data, only a subset of the data needs to be available, rather than all of it, in order for the entire block to be recoverable. We then build a random sampling-based scheme on top of this, to allow light clients to verify with high probability that a subset of the data has been published, to be described in Section 4.5.

Erasure coding and sampling as a potential solution has been briefly discussed on IRC chatrooms with no analysis, however these early ideas [103] require semi-trusted third parties to inform clients of missing pieces of the block, and do not have a means of dealing with block producers that generate invalid codes.

Perard *et al.* [120] have proposed using erasure coding to allow light clients to voluntarily contribute to help storing the blockchain without having to download all of it, however they do not propose a scheme to allow light clients to verify that the full data is available.

RapidChain [157] uses an information dispersal algorithm [124] that uses erasure codes to make block propagation more efficient. Similar to our work, block producers create blocks that commit to the erasure coded version of data, however a data availability proof scheme is not proposed.

Since the release of the paper based on this chapter, new work by Yu *et al.* [156] on data availability proofs has been published that builds on this work, that adopts our security definitions (Definition 1 and Definition 2) and framework. An alternative data availability proof scheme is proposed where only an $O(1)$ hash commitment is required in each header with respect to the size of the block, compared to an $O(\sqrt{n})$ commitment in our scheme. However, this scheme requires light clients to download 2.5-4x more samples from each block to achieve the same level of data availability guarantee [156].

Proofs of Data Possession and Retrievability

There is extensive literature on 'proofs of data possession' (PoDP) [12, 13] and 'proofs of retrievability' (PoR) schemes [33, 78, 133]. PoDPs allow a storage provider (the prover) to prove to a verifier that they have possession of a file [73].

PoRs extend PoDPs by allowing a storage provider to prove to a verifier that they present a protocol interface in which a verifier can retrieve a file in its entirety [78], assuming that the interface remains live. A prover can refuse to release the file (*e.g.,* by closing its protocol interface) after successfully participating in a PoR [78]. However, the acts of proving possession of a file and extracting a file use the same interface, and therefore in theory preventing the extraction of a file would also prevent proving the possession of it.

Our data availability proofs use similar building blocks as PoR schemes. In particular, we employ erasure coding and random sampling to ensure that a file is retrievable with a high probability. However, data availability proofs and PoRs differ in two key ways. Firstly, in PoR schemes the file is chosen by the verifier, and thus can trust themselves to encode the file correctly when it is uploaded to the prover's storage server. This is not the case for data availability proofs, as the file contents is decided by a block producer. Secondly, there is no single authenticated prover in our context, as data availability proofs aim to show that the data has been published to an unauthenticated peer-to-peer network, rather than a single or (specific set of) servers.

4.3 Assumptions and Model

We first present our blockchain model, and then define the core problem solved by fraud and data availability proofs, under this model in Section 4.3.2.

We then present the network and threat model under which our fraud proofs (Section 4.4) and data availability proofs (Section 4.5) apply.

4.3.1 Preliminaries

We present some primitives that we use in the rest of the chapter.

- $\mathsf{hash}(x)$ is a cryptographically secure hash function that returns the digest of x (*e.g.,* SHA-256).

- $\mathsf{root}(L)$ returns the Merkle root for a list of items L.

- $\{e \rightarrow r\}$ denotes a Merkle proof that an element e is a member of the Merkle tree committed by root r.

- $\mathsf{VerifyMerkleProof}(e, \{e \rightarrow r\}, r, n, i)$ returns true if the Merkle proof is valid, otherwise false, where n additionally denotes the total number of elements in the underlying tree and i is the index of e in the tree. This verifies that e is at index i, as well as its membership.

- $\{k, v \rightarrow r\}$ denotes a Merkle proof that a key-value pair k, v is a member of the Sparse Merkle tree committed by root r.

4.3.2 Blockchain Model

We assume a generalised blockchain architecture, where the blockchain consists of a hash-based chain of block headers $H = (h_0, h_1, ...)$. A block header may have state agreement, state validity, or both [158]:

- **State validity.** A block header is considered to have state validity if all the transactions in the block are valid according to the protocol's transaction validity rules.

- **State agreement.** A block header has state agreement if the consensus set is in agreement that it should be included in the chain, *e.g.,* it has $2f+1$ BFT signatures, or it is a part of a longest proof-of-work chain. If the consensus set is dishonest, then a block header that does not have state validity may have state agreement, thus causing light clients to accept invalid blocks.

Each block header h_i contains a Merkle root txRoot_i of a list of transactions T_i, such that $\mathsf{root}(T_i) = \mathsf{txRoot}_i$. Given a node that downloads the list of transactions N_i from the network, a block header h_i is considered to have state validity if (*i*) $\mathsf{root}(N_i) = \mathsf{txRoot}_i$ and (*ii*) given some validity function

$$\mathsf{valid}(T, S) \in \{\mathsf{true}, \mathsf{false}\}$$

where T is a list of transactions and S is the state of the blockchain, then $\mathsf{valid}(T_i, S_{i-1})$ must return true, where S_i is the state of the blockchain after applying all of the transactions in T_i on the state from the previous block S_{i-1}. We assume that $\mathsf{valid}(T,S)$ takes $O(n)$ time to execute, where n is the number of transactions in T.

In terms of transactions, we assume that given a list of transactions $T_i = (t_i^0, t_i^1, ..., t_i^n)$, where t_i^j denotes a transaction j at block i, there exists a state transition function $\mathsf{transition}$ that returns the post-state S' of executing a transaction on a particular pre-state S, or an error if the transition is illegal:

$$\mathsf{transition}(S,t) \in \{S', \mathsf{err}\}$$

$$\mathsf{transition}(\mathsf{err},t) = \mathsf{err}$$

Thus given the intermediate state $I_i^j = \mathsf{transition}(I_i^{j-1}, t_i^j)$ after executing transactions $(t_i^0, t_i^1, ..., t_i^j)$ in block i where $j \leq n$, and the base case $I_i^{-1} = S_{i-1}$, then $S_i = I_i^n$.

Therefore, $\mathsf{valid}(T_i, S_{i-1}) = \mathsf{true}$ if and only if $I_i^n \neq \mathsf{err}$.

Core Problem

How can it be proven to clients that for a given block header h_i, $\mathsf{valid}(T_i, S_{i-i})$ returns false (thus proving that h_i does not have state validity) in less than $O(n)$ time and less than $O(n)$ space, relying on as few security assumptions as possible?

4.3.3 Network and Threat Model

We assume a network that consists of full nodes and light clients as described in Section 2.3.3. As is the status quo in Bitcoin and Ethereum, we assume a network topology as shown in Figure 4.1; full nodes communicate with each other, and light clients communicate with full nodes, but light clients do not communicate with each other.

We outline our threat model in detail below.

Blocks and Consensus

Block headers may be created by adversarial actors, and thus may be invalid, and there is no honest majority of consensus-participating nodes that we can rely on that can confirm state validity.

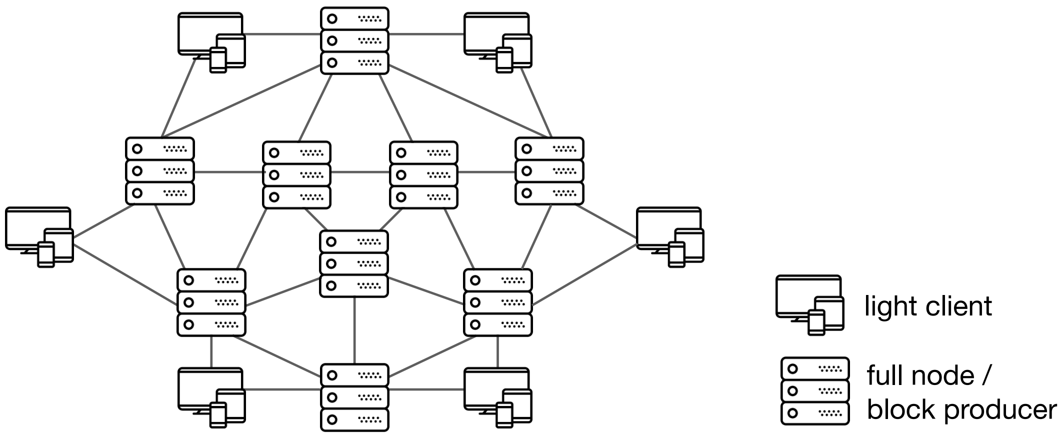


Figure 4.1: Network model—full nodes communicate with each other, and light clients communicate only with full nodes.

Network Synchrony

We assume a synchronous gossiping network [34]. Specifically, we assume a maximum network delay δ; such that if one honest node can connect to the network and download some data (*e.g.,* a block) at time T, then it is guaranteed that any other honest node will be able to do the same at time $T' \leq T + \delta$. In order to guarantee that light clients do not accept block headers that do not have state validity, they must receive fraud proofs in time, hence a synchrony assumption is required.

This synchronous model is a recurring assumption in the consensus protocols of most blockchains [157, 100, 117, 109, 87] due to FLP impossibility [66]. However while full nodes rely on synchrony to determine which block headers have state agreement, they do not rely on synchrony for state validity, as they check the validity of blocks themselves. Standard light clients however do rely on a synchrony assumption for state validity, as they assume that block headers that have state agreement also have state validity. However without synchrony, they cannot know if a block header has state agreement.

Full nodes

Full nodes may be dishonest, *e.g.,* they may not relay information (*e.g.,* fraud proofs), or they may relay invalid blocks. However we assume that the graph of honest full nodes is well connected, a standard assumption made in previous work

[109, 87, 83, 100]. This results in a broadcast network, due to the synchrony assumption made above.

Light clients

As is the status quo in prior work [109, 36], we assume that each light client is connected to at least one honest full node (*i.e.,* is not under an eclipse attack [75]), as this is necessary to achieve a synchronous gossiping network.

However when a light client is connected to multiple full nodes, they do not know which nodes are honest or dishonest, just that at least one of them is. Consequently, light clients may be connected to dishonest full nodes that send block headers that have consensus (state agreement) but correspond to invalid or unavailable blocks (violating state validity), and thus need fraud and data availability proofs to detect this.

For data availability proofs, we assume a minimum number of honest light clients in the network to allow for a block to be reconstructed, as each light client downloads a small chunk of every block. The specific number depends on the parameters of the system, and is analysed in Section 4.5.8.

Double Spending vs. State Invalidity

Our goal is specifically to ensure that light clients do not accept blocks with invalid transactions, in the presence of a dishonest majority of consensus-participating nodes. This is different to double spending attacks, where a dishonest majority forks the chain to undo valid transactions, by breaking consensus finality to change which block headers have state agreement, which is not the focus of this paper. An honest majority assumption is still necessary to prevent double spending attacks for both full nodes and light clients—our goal is to eliminate this assumption for transaction validity, thus significantly limiting the damage that a dishonest majority of consensus-participating nodes can do.

We believe that it is nevertheless vital to prevent dishonest consensus majorities from being able to get invalid transactions accepted, even if double spends are possible. This is because, for example, the electricity cost of executing a 51% attack in Bitcoin for an hour is over 350,000 USD (see Section 6.5.1). If only a

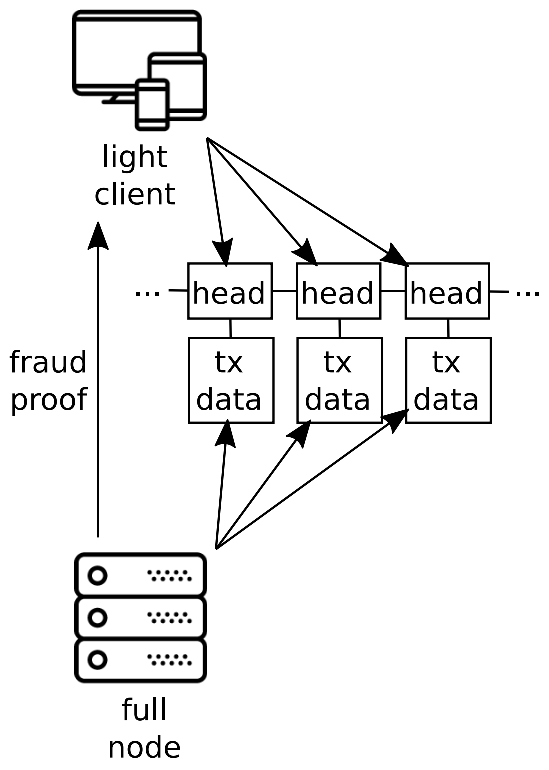


Figure 4.2: Overview of the architecture of a fraud proof system at a network level.

double spend is possible with a 51% attack, the attacker would have to purchase (and receive within one hour) a real-world item that costs at least 350,000 USD to break even (assuming that the attacker already has the hardware, *e.g.,* miners may collude), possibly revealing their identity in the process. However if 51% attacker can get invalid transaction accepted, they could create *e.g.,* transactions that generate unbounded amounts of currency and inflate the monetary supply, which greatly increases the incentive for conducting such an attack.

4.4 Fraud Proofs

4.4.1 Block Structure

In order to support efficient fraud proofs, it is necessary to design a blockchain data structure that supports fraud proof generation by design. Extending the model described in Section 4.3.2, a block header h_i at height i contains the following elements:

$\mathsf{prevHash}_i$ The hash of the previous block header in the chain.

$\mathsf{dataRoot}_i$ The root of the Merkle tree of the data (*e.g.,* transactions) included in the block.

$\mathsf{dataLength}_i$ The number of leaves represented by $\mathsf{dataRoot}_i$.

$\mathsf{stateRoot}_i$ The root of a Sparse Merkle tree of the state of the blockchain (to be described in Section 4.4.2).

$\mathsf{additionalData}_i$ Additional arbitrary data that may be required by the network (*e.g.,* in proof-of-work, this may include a nonce and the target difficulty threshold).

Additionally, the hash of each block header $\mathsf{blockHash}_i = \mathsf{hash}(h_i)$ is also stored by clients and nodes.

Note that typically blockchains have the Merkle root of transactions included in headers. We have abstracted this to a 'Merkle root of data' called $\mathsf{dataRoot}_i$, because as we shall see in Section 4.4.3, as well as including transactions in the block data, we also need to include intermediate state roots.

4.4.2 State Root and Execution Trace Construction

To instantiate a blockchain based on the state-based model described in Section 4.3.2, we make use of Sparse Merkle trees, and represent the state as a key-value map.

The state would need to keep track of all data that is relevant to block processing, including for example the cumulative transaction fees paid to the creator of the current block after each transaction.

We now define a variation of the function $\mathsf{transition}$ defined in Section 4.3.2, called $\mathsf{rootTransition}$, that performs transitions without requiring the whole state tree, but only the state root and Merkle proofs of parts of the state tree that the transaction reads or modifies (which we call "state witness", or w for short). These Merkle proofs are effectively expressed as a sub-tree of the same state tree with a common root.

$$\mathsf{rootTransition}(\mathsf{stateRoot}, t, w) \in \{\mathsf{stateRoot}', \mathsf{err}\}$$

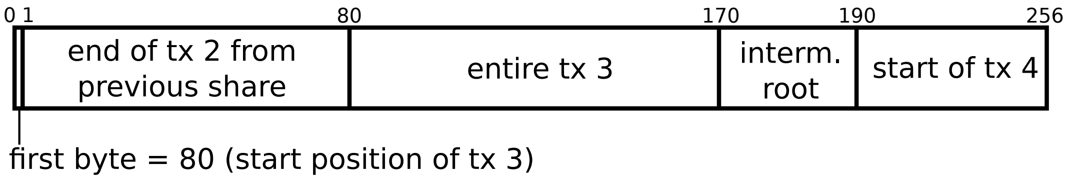

0 1
80
170
190
256
end of tx 2 from previous share
entire tx 3
interm. root
start of tx 4
first byte = 80 (start position of tx 3)

Figure 4.3: Example of a 256-byte share.

A state witness w consists of a set of (k, v) key-value pairs and their associated Sparse Merkle proofs in the state tree, $w = \{(k_0, v_0, \{k_0, v_0 \rightarrow \mathsf{stateRoot}\}), (k_1, v_1, \{k_1, v_1 \rightarrow \mathsf{stateRoot}\}), ...\}$.

After executing t on the parts of the state shown by w, if t modifies any of the state, then the new resulting $\mathsf{stateRoot}'$ can be generated by computing the root of the new sub-tree with the modified leafs. Note that if w is invalid and does not contain all of the parts of the state required by t during execution, then err is returned.

Let us denote, for the list of transactions $T_i = (t_i^0, t_i^1, ..., t_i^n)$, where t_i^j denotes a transaction j at block i, then w_i^j is the state witness for transaction w_i^j for $\mathsf{stateRoot}_i$.

Thus given the intermediate state root $\mathsf{interRoot}_i^j = \mathsf{rootTransition}(\mathsf{interRoot}_i^{j-1}, t_i^j, w_i^j)$ after executing transactions $(t_i^0, t_i^1, ..., t_i^j)$ in block i where $j \leq n$, and the base case $\mathsf{interRoot}_i^{-1} = \mathsf{stateRoot}_{i-1}$, then $\mathsf{stateRoot}_i = \mathsf{interRoot}_i^n$.

4.4.3 Data Root and Periods

The data represented by the $\mathsf{dataRoot}_i$ of a block contains transactions arranged into fixed-size chunks of data called ‘shares’, interspersed with intermediate state roots called ‘traces’ between transactions. We denote trace_i^j as the jth intermediate state root in block i. These intermediate state roots between transactions effectively form an execution trace within the block, forcing the block producer to ‘show’ how it computed the final $\mathsf{stateRoot}_i$ in the block header, so that if the final $\mathsf{stateRoot}_i$ is incorrect, the first incorrect intermediate state root in the execution trace that introduced the error can be pin-pointed. As we will see in Section 4.4.4, it then becomes trivial to efficiently prove to light clients that $\mathsf{stateRoot}_i$ is incorrect, as only the (few) transactions in the execution trace between the last correct intermediate state root and first incorrect intermediate state root need to be shown as proof.

Each leaf in the data tree represents a share. It is necessary to arrange data into fixed-size shares to allow for data availability proofs as we shall see in Section 4.5. Additionally, the reason for interspersing intermediate state roots with transactions in the same tree is because the availability of both the transactions and intermediate state roots need to be guaranteed, as both are required to generate fraud proofs of invalid state transitions as we shall see in Section 4.4.4.

As a share may not contain entire transactions but only parts of transactions as shown in Figure 4.3, we may reserve the first byte in each share to be the starting position of the first transaction that starts in the share, or 0 if no transaction starts in the share. This allows a protocol message parser to establish the message boundaries without needing every transaction in the block.

Given a list of shares $(\mathsf{sh}_0, \mathsf{sh}_1, ...)$ we define a function $\mathsf{parseShares}$ which parses these shares and outputs an ordered list of messages $(m_0, m_1, ...)$, which are either transactions or intermediate state roots. For example, $\mathsf{parseShares}$ on some shares in the middle of some block i may return $(\mathsf{trace}_i^1, t_i^4, t_i^5, t_i^6, \mathsf{trace}_i^2)$.

$$\mathsf{parseShares}((\mathsf{sh}_0, \mathsf{sh}_1, ...)) = (m_0, m_1, ...)$$

Note that as the block data does not necessarily contain an intermediate state root after every transaction, we assume a 'period criterion', a protocol rule that defines how often an intermediate state root should be included in the block's data. For example, the rule could be at least once every p transactions, or b bytes or g gas (*i.e.,* in Ethereum [153]).

We thus define a function $\mathsf{parsePeriod}$ which parses a list of messages, and returns a pre-state intermediate root trace_i^x, a post-state intermediate root trace_i^{x+1}, and a list of transaction $(t_i^g, t_i^{g+1}, ..., t_i^{g+h})$ such that applying these transactions on trace_i^x is expected to return trace_i^{x+1}. If the list of messages violate the period criterion, then the function may return err, for example if there are too many transactions in the messages to constitute a period.

$$\mathsf{parsePeriod}((m_0, m_1, ...)) \in \{(\mathsf{trace}_i^x, \mathsf{trace}_i^{x+1}, (t_i^g, t_i^{g+1}, ..., t_i^{g+h})), \mathsf{err}\}$$

Note that trace_i^x may be *nil* if no pre-state root was parsed, as this may be the case if the first messages in the block are being parsed, and thus the pre-state root is the state root of the previous block $\mathsf{stateRoot}_{i-i}$. Likewise, trace_i^{x+1} may be *nil* if no post-state root was parsed *i.e.,* if the last messages in the block are being parsed, as the post-state root would be $\mathsf{stateRoot}_i$.

4.4.4 Proof of Invalid State Transition

A faulty or malicious miner may provide a bad $\mathsf{stateRoot}_i$ in the block header that modifies the state in an invalid way, *i.e.,* it does not match the new state root that should be returned according to $\mathsf{rootTransition}$. Full nodes can use the execution trace provided in $\mathsf{dataRoot}_i$ to prove to light clients that some part of the execution trace resulting in $\mathsf{stateRoot}_i$ was invalid, by pin-pointing the first intermediate state root that is invalid. This also prevents adversaries from including invalid transactions that cause $\mathsf{rootTransition}$ to return an err symbol, as $\mathsf{stateRoot}_i$ cannot be an err symbol as it is interpreted as a cryptographic hash; the adversary would simply have to provide an invalid arbitrary state root.

We define a function $\mathsf{VerifyTransitionFraudProof}$ and its parameters which verifies fraud proofs of invalid state transitions received from full nodes. We denote d_i^j as share number j in block i.

Summary of VerifyTransitionFraudProof. A state transition fraud proof consists of (*i*) the relevant shares in the block that contain the bad state transition, (*ii*) Merkle proofs that those shares are in $\mathsf{dataRoot}_i$, and (*iii*) the state witnesses for the transactions contained in those shares. The function takes as input this fraud proof, then (*i*) verifies the Merkle proofs of the shares, (*ii*) parses the transactions from the shares, and (*iii*) checks if applying the transactions on the intermediate pre-state root results in the intermediate post-state root specified in the shares. If it does not, then the fraud proof is valid, and the block that the fraud proof is for should be permanently rejected by the client.

Importantly, the client does not need to know the entire state of the blockchain to validate this fraud proof; clients only need to know the relevant parts of the state that the relevant transactions access, which are provided by the state witnesses.

$$\begin{aligned}
&\mathsf{VerifyTransitionFraudProof}(\mathsf{blockHash}_i, \\
&\quad (d_i^y, d_i^{y+1}, \ldots), y, && \text{(shares)} \\
&\quad (\{d_i^y \to \mathsf{dataRoot}_i\}, \{d_i^{y+1} \to \mathsf{dataRoot}_i\}, \ldots), && \text{(share Merkle proofs)} \\
&\quad (w_i^g, w_i^{g+1}, \ldots, w_i^{g+h}) && \text{(state witnesses)} \\
&) \in \{\mathsf{true}, \mathsf{false}\}
\end{aligned}$$

VerifyTransitionFraudProof returns true if all of the following conditions are met, otherwise false is returned:

1. $\mathsf{blockHash}_i$ corresponds to a block header h_i that the client has downloaded and stored.

2. For each share d_i^{y+a} in the proof, $\mathsf{VerifyMerkleProof}(d_i^{y+a}, \{d_i^{y+a} \to \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, \mathsf{dataLength}_i, y+a)$ must return true.

3. Given $\mathsf{parsePeriod}(\mathsf{parseShares}((d_i^y, d_i^{y+1}, \ldots))) \in \{(\mathsf{trace}_i^x, \mathsf{trace}_i^{x+1}, (t_i^g, t_i^{g+1}, \ldots, t_i^{g+h})), \mathsf{err}\}$, the result must not be err. If trace_i^x is *nil*, then $y = 0$ must be true, and if trace_i^{x+1} is *nil*, then $y + m = \mathsf{dataLength}_i$ must be true.

4. Check that applying $(t_i^g, t_i^{g+1}, \ldots, t_i^{g+h})$ on trace_i^x results in trace_i^{x+1}. Formally, let the intermediate state root after executing transactions $(t_i^0, t_i^1, \ldots, t_i^j)$ in block i be $\mathsf{interRoot}_i^j = \mathsf{rootTransition}(\mathsf{interRoot}_i^{j-1}, t_i^j, w_i^j)$. If trace_i^x is not *nil*, then the base case is $\mathsf{interRoot}_i^y = \mathsf{trace}_i^x$, otherwise $\mathsf{interRoot}_i^y = \mathsf{stateRoot}_{i-1}$. If trace_i^{x+1} is not *nil*, $\mathsf{trace}_i^{x+1} = \mathsf{interRoot}_i^{g+h}$ must be true, otherwise $\mathsf{stateRoot}_i = \mathsf{interRoot}_i^{y+m}$ must be true.[1]

4.4.5 Transaction Fees

As discussed in Section 4.4.2, the state would need to keep track of all data that is relevant to block processing. A block producer may attempt to collect more

[1]For simplicity, we assume a model where state witnesses are provided for every individual intermediate state root within the trace, but it is also possible to only provide state witnesses only for the trace intermediate pre-state root, and execute the transactions as a single batch.

transaction fees than is afforded to them by the transactions in the block. In order to make this detectable by a fraud proof as part of the model we have described, we can introduce a special key in the state tree called `__fees__`, which represents the cumulative fees in the block after applying each transaction, and is reset to 0 after applying the transaction where the block producer collects the fees.

4.5 Data Availability Proofs

A malicious block producer could prevent full nodes from generating fraud proofs by withholding the data needed to recompute $\mathsf{dataRoot}_i$ and only releasing the block header h_i to the network. The block producer could then only release the data—which may contain invalid transactions or state transitions—long after the block has been published, and make the block invalid. This would cause a rollback of transactions on the ledger of future blocks. It is therefore necessary for light clients to have a level of assurance that the data matching $\mathsf{dataRoot}_i$ is indeed available to the network.

We propose a data availability scheme based on Reed-Solomon erasure coding, where light clients request random shares of data to get high probability guarantees that all the data associated with the root of a Merkle tree is available. The scheme assumes there is a sufficient number of honest light clients making the same requests such that the network can recover the data, as light clients upload these shares to full nodes, if a full node that does not have the complete data requests it. It is fundamental for light clients to have assurance that all the transaction data is available, because it is only necessary to withhold a few bytes to hide an invalid transaction in a block.

We define below *soundness* and *agreement* and analyse them in Section 4.5.9. We use k as a constant that depends on the size and topology of the peer-to-peer network.

Definition 1 (Soundness). *If an honest light client accepts a block as available, then at least one honest full node has the full block data or will have the full block data within some known maximum delay $k \cdot \delta$ where δ is the maximum network delay.*

Definition 2 (Agreement)**.** *If an honest light client accepts a block as available, then all other honest light clients will accept that block as available within some known maximum delay $k \cdot \delta$ where δ is the maximum network delay.*

4.5.1 Background on Erasure Codes and Reed-Solomon Codes

Erasure codes are error-correcting codes [60, 121] working under the assumption of bit erasures rather than bit errors; in particular, the users knows which bits have to be reconstructed. Error-correcting codes transform a message of length k into a longer message of length $n > k$ such that the original message can be recovered from a subset of the n shares.

Reed-Solomon (RS) codes [150] have various applications and are among the most studied error-correcting codes. A Reed-Solomon code encodes data by treating a length-k message as a list of elements $x_0, x_1, ..., x_{k-1}$ in some finite field (prime fields and binary fields are most frequently used), interpolating the polynomial $P(x)$ where $P(i) = x_i$ for all $0 \leq i < k$, and then extending the list with $x_k, x_{k+1}, ..., x_{n-1}$ where $x_i = P(i)$. The polynomial P can be recovered from any k shares from this longer list using techniques such as Lagrange interpolation, or more optimized and advanced techniques involving tools such as Fast Fourier transforms, and knowing P one can then recover the original message. Reed-Solomon codes can detect and correct any combination of up to $\frac{n-k}{2}$ errors, or combinations of errors and erasures. RS codes have been generalised to multidimensional codes [134, 57] in various ways [135, 154, 130]. In a d-dimensional code, the message is encoded into a square or cube or hybercube of size $k \times k \times ... \times k$, and a multidimensional polynomial $P(x_1, x_2, ..., x_d)$ is interpolated where $P(i_1, i_2, ..., i_n) = x_{i_1, i_2 ..., i_n}$, and this polynomial is extended to a larger $n \times n \times ... \times n$ square or cube or hypercube.

4.5.2 Strawman 1D Reed-Solomon Availability Scheme

To provide some intuition, we first describe a strawman data availability scheme, based on standard Reed-Solomon coding.

A block producer compiles a block of data consisting of k shares, extends the data to $2k$ shares using Reed-Solomon encoding, and computes a Merkle root (the

$\mathsf{dataRoot}_i$) over the extended data, where each leaf corresponds to one share.

When light clients receive a block header with this $\mathsf{dataRoot}_i$, they randomly sample from full nodes shares from the Merkle tree that $\mathsf{dataRoot}_i$ represents, and only accept a block once it has received all of the shares requested. If an adversarial block producer makes more than 50% of the shares unavailable to make the full data unrecoverable (recall in Section 4.5.1 that Reed-Solomon codes allow recovery of $2t$ shares from any t shares), there is a 50% chance that a client will randomly sample an unavailable share in the first draw and thus detect that the data is unavailable, a 75% chance after two draws, a 87.5% chance after three draws, and so on, if they draw with replacement. (In the full scheme, they will draw without replacement, and so the probability will be even higher.)

Note that for this scheme to work, there must be enough light clients in the network sampling enough shares so that block producers will be required to release more than 50% of the shares in order to pass the sampling challenge of all light clients, and so that the full block can be recovered. An in-depth probability and security analysis is provided in Section 4.5.8.

The problem with this scheme is that an adversarial block producer may incorrectly compute the extended Reed-Solomon encoded data, and thus the incomplete block is unrecoverable from the extended data even if more than 50% of the data is available. With standard Reed-Solomon encoding, the fraud proof that the extended data is invalid is the original data itself, as clients would have to re-encode all data locally to verify the mismatch with the given extended data, and thus it requires $O(n)$ data with respect to the size of the block. Therefore, we instead use multi-dimensional encoding, as will be described in Section 4.5.3, so that proofs of incorrectly generated codes are limited to a specific axis—rather than the entire data—reducing the fraud proof size to $O(\sqrt[d]{n})$ where d is the number of dimensions of the encoding. For simplicity, we will only consider two-dimensional Reed-Solomon encoding in this paper, but our scheme can be generalised to higher dimensions.

In the sections below, we proceed to describe the details of the full data avail-

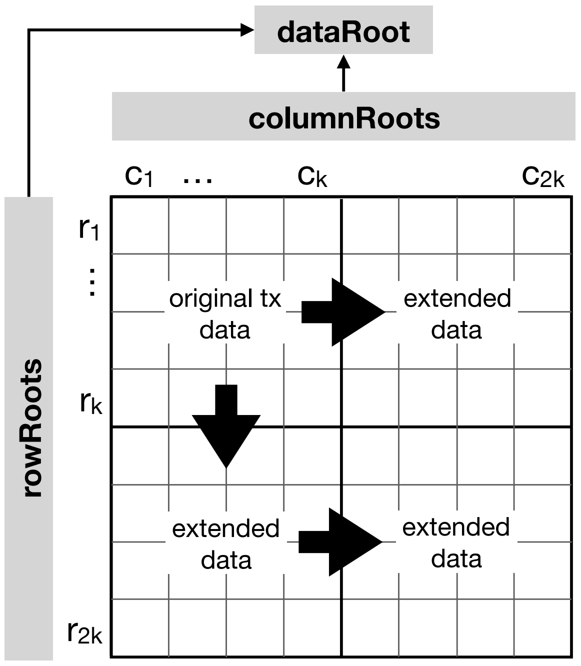


Figure 4.4: Diagram showing a 2D Reed-Solomon encoding. The original data is initially arranged in a $k \times k$ matrix, which is then 'extended' to a $2k \times 2k$ matrix applying multiple times Reed-Solomon encoding.

ability scheme.

4.5.3 2D Reed-Solomon Encoded Merkle Tree Construction

In this section we describe how to compute a $\mathsf{dataRoot}_i$ for block header i using a 2D Reed-Solomon code.

Let extend be a function that takes in a list of k shares, and returns a list of $2k$ shares that represent the extended shares encoded using a standard 1D Reed-Solomon code.

$$\mathsf{extend}(\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_k) = (\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_{2k})$$

The first k shares that are returned are the same as the input shares, and the latter k are the coded shares. Recall that all $2k$ shares can be recovered with knowledge of any k of the $2k$ shares.

A 2D Reed-Solomon Encoded Merkle tree can then be constructed as follows from a block of data:

1. Split the original data into shares of size $\mathsf{shareSize}$ each, and arrange them into a $k \times k$ matrix O_i; apply padding if the last share is not exactly of size $\mathsf{shareSize}$, or if there are not enough shares to complete the matrix. In the next step, we extend this $k \times k$ matrix to a $2k \times 2k$ matrix M_i with Reed-Solomon encoding.

2. For each row in the original $k \times k$ matrix O_i, pass the k shares in that row to $\mathsf{extend}(\mathsf{sh}_1, \mathsf{sh}_2, \dots, \mathsf{sh}_k)$ and append the extra shares outputted $(\mathsf{sh}_{k+1}, \dots, \mathsf{sh}_{2k})$ to the row to create an extended row of length $2k$, thus extending the matrix horizontally. Repeat this process for the columns in O_i to extend the matrix vertically, so that each original column now has length $2k$. This creates an extended $2k \times 2k$ matrix with the upper-right and lower-left quadrants filled, as shown in Figure 4.4. Then finally apply Reed-Solomon encoding horizontally on each row of the vertically extended portion of the matrix to complete the bottom-right quadrant of the $2k \times 2k$ matrix. This results in the extended matrix M_i for block i.

3. Compute the root of the Merkle tree for each row and column in the $2k \times 2k$ matrix, where each leaf is a share. We have $\mathsf{rowRoot}_i^j = \mathsf{root}((M_i^{j,1}, M_i^{j,2}, \dots, M_i^{j,2k}))$ and $\mathsf{columnRoot}_i^j = \mathsf{root}((M_i^{1,j}, M_i^{2,j}, \dots, M_i^{2k,j}))$, where $M_i^{x,y}$ represents the share in row x, column y in the matrix.

4. Compute the root of the Merkle tree of the roots computed in step 3 and use this as $\mathsf{dataRoot}_i$. We have $\mathsf{dataRoot}_i = \mathsf{root}((\mathsf{rowRoot}_i^1, \mathsf{rowRoot}_i^2, \dots, \mathsf{rowRoot}_i^{2k}, \mathsf{columnRoot}_i^1, \mathsf{columnRoot}_i^2, \dots, \mathsf{columnRoot}_i^{2k}))$.

We note that in step 2, we have chosen to extend the vertically extended portion of the matrix horizontally to complete the extended matrix, however it would also be just as fine to extend the horizontally extended portion of the matrix vertically to complete the extended matrix; this will result in the same matrix because Reed-Solomon coding is linear and commutative with itself [134]. The resulting matrix has the property that all rows and columns have reconstruction capabilities, even columns $C_{k+1}, \dots, C_{2k}$.

The resulting tree of $\mathsf{dataRoot}_i$ has $\mathsf{dataLength}_i = 2 \times (2k)^2$ elements, where the first $\frac{1}{2}\mathsf{dataLength}_i$ elements are in leaves via the row roots, and the latter half are in leaves via the column roots.

Note that although it is possible to present a Merkle proof from $\mathsf{dataRoot}_i$ to an individual share, it is important to note that a Merkle tree has 2^x leaves, and the Merkle sub-trees for the row and column roots are constructed independently from $\mathsf{dataRoot}_i$. Therefore it is necessary to have a wrapper function around $\mathsf{VerifyMerkleProof}$ called $\mathsf{VerifyShareMerkleProof}$ with the same parameters which takes into account how the underlying Merkle tree deals with an unbalanced number of leaves; this may involve calling $\mathsf{VerifyMerkleProof}$ twice for different portions of the path, or offsetting the index.

The width of the matrix can be derived as $\mathsf{matrixWidth}_i = \sqrt{\frac{1}{2}\mathsf{dataLength}_i}$. If we are only interested in the row and column roots of $\mathsf{dataRoot}_i$, rather than the actual shares, then we can assume that $\mathsf{dataRoot}_i$ has $2 \times \mathsf{matrixWidth}_i$ leaves when verifying a Merkle proof of a row or column root.

A light client or full node is able to reconstruct $\mathsf{dataRoot}_i$ from all the row and column roots by recomputing step 4. In order to gain data availability assurances, all light clients should at minimum download all the row and column roots needed to reconstruct $\mathsf{dataRoot}_i$ and check that step 4 was computed correctly, because as we shall see in Section 4.5.7, they are necessary to generate fraud proofs of incorrectly generated extended data.

We nevertheless represent all of the row and column roots as a a single $\mathsf{dataRoot}_i$ to allow 'super-light' clients which do not download the row and column roots, but these clients cannot be assured of data availability and thus do not fully benefit from the increased security of allowing fraud proofs.

Alternative codes and design space

In our system we have chosen to use Reed-Solomon codes with an equal ratio of message to recovery bits. Although other ratios may be used, we parameterise our protocol and security analysis with this ratio for the sake of simplicity, and leave exploration of different parameters for future work.

In terms of the choice of code used, we require two key properties. Firstly, there must be a way to efficiently verify that the code was constructed incorrectly. We have made use of a two-dimensional code, so that a fraud proof of incorrect code is limited to only rows and columns, rather than the entire code. In order to construct a two-dimensional code, the underlying coding scheme must be linear and commutative with itself [134]. Reed-Solomon codes are the most widespread and among the most efficient codes that have this property.

Secondly, it must be possible to authenticate that the shares of a code are generated by the block producer, without interacting directly with the block producer. This is because as discussed in our network model (Section 4.3.3), blocks are propagated in a peer-to-peer network, therefore nodes may not have a direct connection to the block producer. This makes rateless codes such as LT codes [99] unsuitable, because they have an unbounded number of shares, and thus it is not possible for the block producer to commit to all the shares using a Merkle root.

Falcon codes [79] is a construction which allows for authenticated LT codes, however it relies on the assumption that the sender and receiver of the codes have a shared secret key, which would require direct interaction between the block producer and nodes. Krohn *et al.* propose a construction for cryptographically verifying the correctness of rateless codes in a peer-to-peer setting [88]. However, it is not fully secure against a malicious encoder that produces correctly coded shares (due to 'distribution attacks' [88]).

4.5.4 Random Sampling and Network Block Recovery

In order for any share in the 2D Reed-Solomon matrix to be unrecoverable, then at least $(k+1)^2$ out of $(2k)^2$ shares must be unavailable (see Theorem 7), as opposed to $k+1$ out of $2k$ with a 1D code (*i.e.,* more than 50% as mentioned in the strawman scheme in Section 4.5.2). When light clients receive a new block header from the network, they should randomly sample $0 < s \leq (2k)^2$ distinct shares from the extended matrix, and only accept the block if they receive all shares. The higher the s, the greater the confidence a light client can have that the data is available (this will be analysed in Section 4.5.8), just as mentioned in the strawman scheme in Section 4.5.2. Additionally, light clients gossip shares that they have received to the network, so that the full block can be recovered by honest full nodes.

The protocol between a light client and the full nodes that it is connected to works as follows:

1. The light client receives a new block header h_i from one of the full nodes it is connected to, and a set of row and column roots $R = (\mathsf{rowRoot}_i^1, \mathsf{rowRoot}_i^2, ..., \mathsf{rowRoot}_i^{2k}, \mathsf{columnRoot}_i^1, \mathsf{columnRoot}_i^2, ..., \mathsf{columnRoot}_i^{2k})$. If the check $\mathsf{root}(R) = \mathsf{dataRoot}_i$ is false, then the light client rejects the header.

2. The light client randomly chooses a set of unique (x, y) coordinates $S = \{(x_0, y_0)(x_1, y_1), ..., (x_n, y_n)\}$ where $0 < x \leq \mathsf{matrixWidth}_i$ and $0 < y \leq \mathsf{matrixWidth}_i$, corresponding to points on the extended matrix, and sends them to one or more of the full nodes it is connected to.

3. If a full node has all of the shares corresponding to the coordinates in S and their associated Merkle proofs, then for each coordinate (x_a, y_b) the full node responds with $M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{rowRoot}_i^a\}$ or $M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{columnRoot}_i^b\}$. Note that there are two possible Merkle proofs for each share; one from the row roots, and one from the column roots, and thus the full node must also specify for each Merkle proof if it is associated with a row or column root.

4. For each share $M_i^{x_a,y_b}$ that the light client has received, the light client checks $\mathsf{VerifyMerkleProof}(M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{rowRoot}_i^a\}, \mathsf{rowRoot}_i^a, \mathsf{matrixWidth}_i, b)$ is true if the proof is from a row root, otherwise if the proof is from a column root then $\mathsf{VerifyMerkleProof}(M_i^{x_a,y_b}, \{M_i^{x_a,y_b} \rightarrow \mathsf{columnRoot}_i^b\}, \mathsf{columnRoot}_i^b, \mathsf{matrixWidth}_i, a)$ is true.

5. Each share and valid Merkle proof that is received by the light client is gossiped to all the full nodes that the light client is connected to if the full nodes do not have them, and those full nodes gossip it to all of the full nodes that they are connected to.

6. If all the proofs in step 4 succeeded, and no shares are missing from the sample made in step 2, then the block is accepted as available if within $2 \times \delta$ no fraud proofs for the block's erasure code is received (to be described in Section 4.5.7).

4.5.5 Extraction Model

We describe an extraction model, namely an interactive algorithm that governs the recovery of a block. In the classical extraction model in PoR schemes [78, 133, 28, 13, 148], the *extractor* aims to recover a specific block from a storage server (the prover). In our data availability proofs scheme however, the block data is not recovered from a specific server, but from the peer-to-peer network from nodes that have shares of data from blocks. Another fundamental difference with the classical extraction model where the verifier of the PoR is also the extractor: in our scheme, only full nodes act as extractors—not light clients. Light clients verify data availability proofs in order to ensure that full nodes are able to extract the block, and thus generate fraud proofs.

Given a full node that wants to recover a matrix M_i associated with block i, the extraction process proceeds as follows:

1. The full node picks a set of random shares that it does not have, and samples them from one or more of the full nodes it is connected to, using the same random sampling protocol in Section 4.5.4.

2. If as a result of downloading any new share, the row or column that the share is in has greater than $k+1$ recovered shares, then recover the whole row and/or column with recover (see Section 4.5.7).

3. If as a result of the previous step, any incomplete row or column in M_i has greater than $k+1$ recovered shares, then recover the whole row or column with recover. Repeat this step until M_i does not change and no new rows or columns are recoverable.

4. Repeat from step 1 until the whole matrix is recovered.

Note that in step 5 in Section 4.5.4, it is also mentioned that light clients gossip shares to full nodes that do not have them. We note that the share gossiping mechanism among full nodes can be generalised as a peer-to-peer file-sharing network such as BitTorrent or IPFS [47, 25], to enable full nodes to download shares that they do not have, as an alternative to step 1.

Assumption on the Query Interface

Extraction algorithms in PoR schemes work under the assumption that the query interface for querying shares is live, and that the prover replies to the extractor. In our system, the query interface is effectively the entire peer-to-peer network. Specifically, our extraction mechanism assumes that once the shares in a block are published and available, then there will be a copy of at least $(2k)^2 - (k+1)^2$ shares that can be downloaded from the network, so that the block can be recovered. If at any point this no longer is the case, the block cannot be extracted. For our fraud proofs use case, we only require the block to be extractable within the maximum delay $k \times \delta$ after the block is published, so that a fraud proof can be generated.

4.5.6 Selective Share Disclosure

If a block producer selectively releases shares as light clients ask for them, up to $(2k)^2 - (k+1)^2$ shares, they can violate the soundness property (Definition 1) of the clients that ask for the first $(2k)^2 - (k+1)^2$ out of $(2k)^2$ shares, as they will accept the blocks as available despite them being unrecoverable; recall that if $(k+1)^2$ shares are unavailable, the Reed-Solomon matrix may be unrecoverable.

This can be alleviated if one assumes an enhanced network model where a sufficient number of honest light clients make requests such that more than $(2k)^2 - (k+1)^2$ shares will be sampled, and that each sample request for each share is anonymous (*i.e.,* sample requests cannot be linked to the same client) and the distribution in which every sample request is received is uniformly random, for example by using a mix net [45]. As the network would not be able to link different per-share sample requests to the same clients, shares cannot be selectively released on a per-client basis.

We thus assume two network connection models that sample requests can be made under, which we analyse in Section 4.5.9:

- **Standard model.** Sample requests are linkable to the clients that made them, and the order that they are received is predictable (*e.g.,* they are received in the order that they were sent).
- **Enhanced model.** Different sample requests cannot be linked to the same client, and the order that they are received by the network is uniformly random with respect to other requests.

Whether the standard model is acceptable depends on the size of the network and threat model; we will see from Corollary 1 that under reasonable parameters only up to a few hundred clients can be fooled. In a network with hundreds of thousands of users for example (a popular Bitcoin SPV wallet for Android has millions of installs[2]), only small but expensive targeted attacks would be possible.

4.5.7 Fraud Proofs of Incorrectly Generated Extended Data

If a full node has enough shares to recover a particular row or column, and after doing so detects that recovered data does not match its respective row or column root, then it should distribute a fraud proof consisting of enough shares in that row or column to be able to recover it, and a Merkle proof for each share.

We define a function VerifyCodecFraudProof and its parameters that verifies these fraud proofs, where $\mathsf{axisRoot}_i^j \in \{\mathsf{rowRoot}_i^j, \mathsf{columnRoot}_i^j\}$. These proofs can also be verified by 'super-light' clients as they do not assume any knowledge of the row and column roots. We denote axis and ax_j as row or column boolean indicators; 0 for rows and 1 for columns.

Summary of VerifyCodecFraudProof. The fraud proof consists of (*i*) the Merkle root of the incorrectly generated row or column, (*ii*) a Merkle proof that the row or column root is in the data tree, (*iii*) enough shares to be able to reconstruct that row or column, and (*iv*) Merkle proofs that each share is in the data tree. The

[2]`https://play.google.com/store/apps/details?id=de.schildbach.wallet`

function takes as input this fraud proof, and checks that (*i*) all of the supplied Merkle proofs are valid, (*ii*) all of the shares given by the prover are in the same row or column and (*iii*) that the recovered row or column indeed does not match the row or column root in the block. If all these conditions are true, then the fraud proof is valid, and the block that the fraud proof is for should be permanently rejected by the client.

$$\begin{aligned}
&\mathsf{VerifyCodecFraudProof}(\mathsf{blockHash}_i,\\
&\quad \mathsf{axisRoot}_i^j, \{\mathsf{axisRoot}_i^j \rightarrow \mathsf{dataRoot}_i\}, j, && \text{(row or column root)}\\
&\quad \mathsf{axis}, && \text{(row or column indicator)}\\
&\quad ((\mathsf{sh}_1, \mathsf{pos}_1, \mathsf{ax}_1), (\mathsf{sh}_2, \mathsf{pos}_2, \mathsf{ax}_2), ..., (\mathsf{sh}_k, \mathsf{pos}_k, \mathsf{ax}_k)), && \text{(shares)}\\
&\quad (\{\mathsf{sh}_1 \rightarrow \mathsf{dataRoot}_i\}, \{\mathsf{sh}_2 \rightarrow \mathsf{dataRoot}_i\}, ..., \{\mathsf{sh}_k \rightarrow \mathsf{dataRoot}_i\})\\
&) \in \{\mathsf{true}, \mathsf{false}\}
\end{aligned}$$

Let recover be a function that takes a list of shares and their positions in the row or column $((\mathsf{sh}_1, \mathsf{pos}_1), (\mathsf{sh}_2, \mathsf{pos}_2), ..., (\mathsf{sh}_k, \mathsf{pos}_k))$, and the length of the original row or column k. The function outputs the full recovered shares $(\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_{2k})$ or err if the shares are unrecoverable.

$$\mathsf{recover}(((\mathsf{sh}_1, \mathsf{pos}_1), (\mathsf{sh}_2, \mathsf{pos}_2), ..., (\mathsf{sh}_k, \mathsf{pos}_k)), k) \in \{(\mathsf{sh}_1, \mathsf{sh}_2, ..., \mathsf{sh}_{2k}), \mathsf{err}\}$$

$\mathsf{VerifyCodecFraudProof}$ returns true if all of the following conditions are met:

1. $\mathsf{blockHash}_i$ corresponds to a block header h_i that the client has downloaded and stored.

2. If $\mathsf{axis} = 0$ (row root), $\mathsf{VerifyMerkleProof}(\mathsf{axisRoot}_i^j, \{\mathsf{axisRoot}_i^j \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, 2 \times \mathsf{matrixWidth}_i, j)$ returns true.

3. If $\mathsf{axis} = 1$ (col. root), $\mathsf{VerifyMerkleProof}(\mathsf{axisRoot}_i^j, \{\mathsf{axisRoot}_i^j \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, 2 \times \mathsf{matrixWidth}_i, \frac{1}{2}\mathsf{dataLength}_i + j)$ returns true.

4. For each $(\mathsf{sh}_x, \mathsf{pos}_x, \mathsf{ax}_x)$, $\mathsf{VerifyShareMerkleProof}(\mathsf{sh}_x, \{\mathsf{sh}_x \rightarrow \mathsf{dataRoot}_i\}, \mathsf{dataRoot}_i, \mathsf{dataLength}_i, \mathsf{index})$ returns true, where index is the expected index of the sh_x in the data tree based on pos_x assuming it is in the same row or column as $\mathsf{axisRoot}_i^j$. See Appendix A.1 for how index can be computed.

   Note that full nodes can specify Merkle proofs of shares in rows or columns from either the row or column roots *e.g.,* if a row is invalid but the full nodes only has Merkle proofs for the row's share from column roots. This also allows for full nodes to generate fraud proofs if there are inconsistencies in the data between rows and columns *e.g.,* if the same cell in the matrix has a different share in its row and column trees.

5. $\mathsf{root}(\mathsf{recover}(((\mathsf{sh}_1, \mathsf{pos}_1), (\mathsf{sh}_2, \mathsf{pos}_2), ..., (\mathsf{sh}_k, \mathsf{pos}_k)), k)) = \mathsf{axisRoot}_i^j$ is false.

4.5.8 Sampling Security Analysis

We present how the data availability scheme presented in Section 4.5 can provide lights clients with a high level of assurance that block data is available to the network.

Minimum Unavailable Shares for Unrecoverability

Theorem 7 states that data is unrecoverable if a malicious block proposer withholds $k+1$ shares of at least $k+1$ columns or rows; which makes a total of $(k+1)^2$ minimum shares to withhold.

Theorem 7. *Given a $2k \times 2k$ extended matrix E as show in Figure 4.4, data is unrecoverable if at least $k+1$ columns or rows have each at least $k+1$ unavailable shares. In that case, the minimum number of shares that must be unavailable is $(k+1)^2$.*

Proof. Suppose a malicious block producer wants to make unrecoverable a share $E_{i,j}$ of the $2k \times 2k$ matrix E. Recall that Reed-Solomon encoding allows recovery of all $2k$ shares from any k shares; the block producer will have to (*i*) make unavailable at least $k+1$ shares from the row $E_{i,*}$, and (*ii*) make unavailable at least $k+1$ shares from the column $E_{*,j}$.

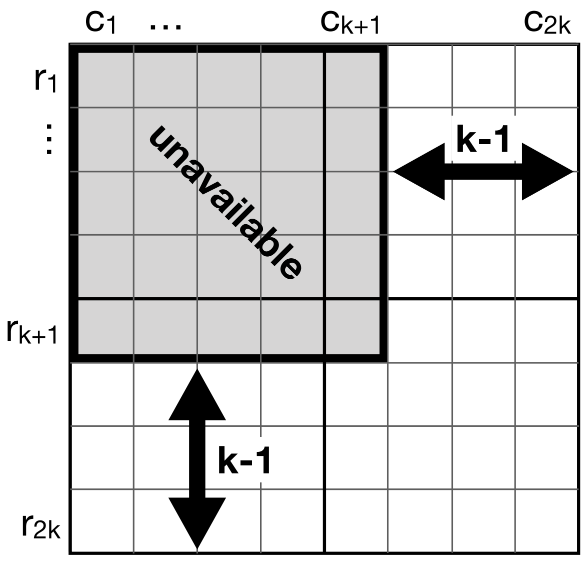


Figure 4.5: Graphical interpretation of Theorem 7. Data is unrecoverable if at least $k+1$ columns (or rows) have each at least $k+1$ unavailable shares.

Let us start from (*i*); the block producer withholds at least $k+1$ shares from row $E_{i,*}$. However, each of these $k+1$ withheld shares $(E_{i,c_1},\dots,E_{i,c_{k+1}}) \in E_{i,*}$ can be recovered from the available shares of their respective columns $E_{*,c_1},E_{*,c_2},\dots,E_{*,c_{k+1}}$. Therefore, the block producer will also have to withhold at least $k+1$ shares from each of these columns. This gives a total of $(k+1)*(k+1) = (k+1)^2$ shares to withhold. Note that at this point, there are not enough shares left in the matrix to recover any of the $(k+1)^2$ shares of columns $E_{*,c_1},\dots,E_{*,c_{k+1}}$.

Let us now consider (*ii*); the block producer withholds at least $k+1$ shares from the column $E_{*,j}$ to make unrecoverable the share $E_{i,j}$. As before, each shares $(E_{r_1,j},\dots,E_{r_{k+1},j}) \in E_{*,j}$ can be recovered from the available shares of their respective row $E_{r_1,*},E_{r_2,*},\dots,E_{r_{k+1},*}$. Therefore, the block producer will also have to withhold at least at least $k+1$ shares from each of these rows. As before, this also gives a total of $(k+1)*(k+1) = (k+1)^2$ shares to withhold.

However, (*i*) is equivalent to (*ii*) by the symmetry of the matrix, and are actually operating on the same shares; executing (*i*) on matrix E is equivalent to executing (*ii*) on the transpose of matrix E. □

We note that Theorem 7 relies on a necessary but not sufficient condition for an adversary to cause a block to be unrecoverable; $(k+1)^2$ is the *minimum* number of shares that need to be withheld. In particular, if the adversary does not withhold the exact correct $(k+1)^2$ shares to form a $(k+1)\times(k+1)$ square, then the matrix may be recoverable. This means that in practice the adversary will likely have to withhold more than $(k+1)^2$ shares because it is unlikely that clients will randomly sample shares in a perfect square; thus we assume a stronger adversary than likely exists in practice for our security analysis. Therefore the analysis is conservative in terms of security.

Unrecoverable Block Detection

Theorem 8 states the probability that a single light client will sample at least one unavailable share in a matrix with the minimum unavailable shares for unrecoverability, thus detecting that a block may be unrecoverable.

Theorem 8. *Given a $2k\times 2k$ extended matrix E as shown in Figure 4.4, where $(k+1)^2$ shares are unavailable. If one player randomly samples $0 < s \leq (2k)^2$ shares from E, the probability of sampling at least one unavailable share is:*

$$p_1(X \geq 1) = 1 - \prod_{i=0}^{s-1}\left(1 - \frac{(k+1)^2}{4k^2 - i}\right) \tag{4.1}$$

Proof. We start by assuming that the $2k\times 2k$ matrix E contains q unavailable shares; If the player performs s trials $(0 < s \leq (2k)^2)$, the probability of finding exactly zero unavailable shares is:

$$p_1(X = 0) = \frac{\binom{4k^2-q}{s}}{\binom{4k^2}{s}} \tag{4.2}$$

The numerator of Equation (4.2) computes the number of ways to pick s shares among the set of available shares $4k^2 - q$ (i.e., $\binom{4k^2-q}{s}$). The denominator computes the total number of ways to pick any s shares out of the total number of shares (i.e., $\binom{4k^2}{s}$).

Then, the probability $p_1(X \geq 1)$ of finding at least one unavailable share can

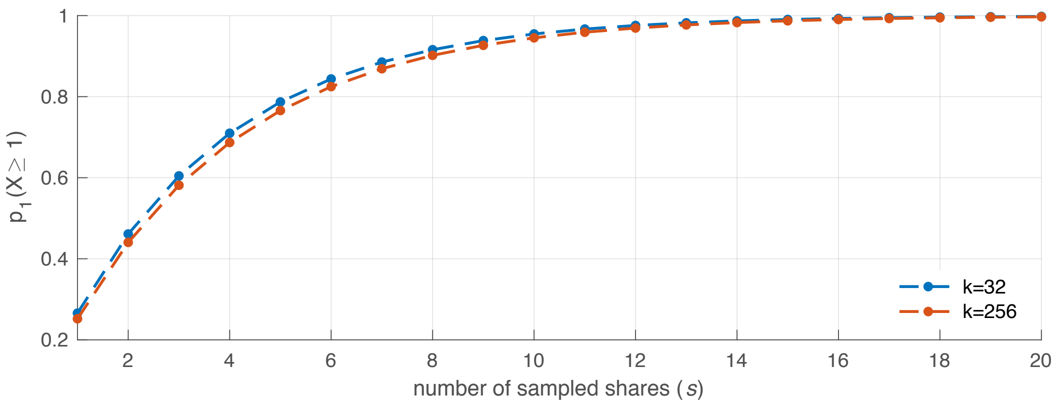


Figure 4.6: Plot of Equation (4.1)—variation of the probability $p_1(X \geq 1)$ with the number of sampled shares (s) (computed for $k = 32$ and $k = 256$).

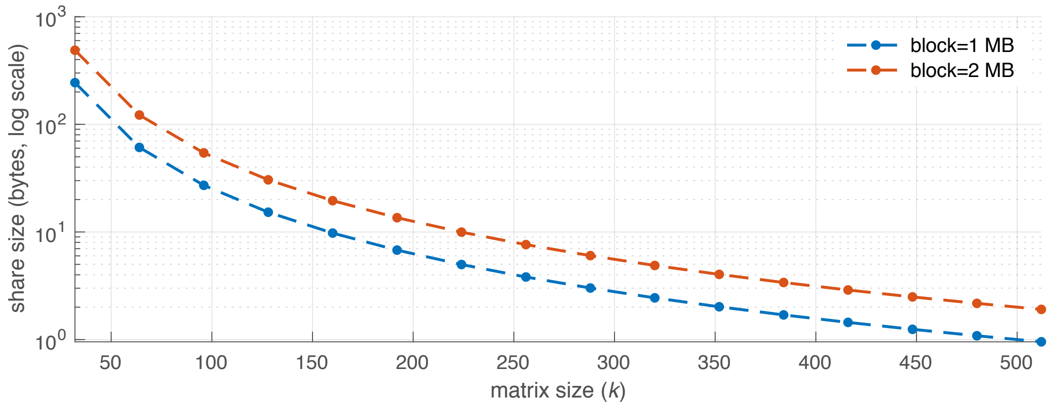


Figure 4.7: Variation of the shares size with the size of the matrix (k).

be easily computed from Equation (4.2):

$$\begin{aligned} p_1(X \geq 1) &= 1 - p_1(X = 0) && (4.3) \\ &= 1 - \frac{\binom{4k^2-q}{s}}{\binom{4k^2}{s}} && (4.4) \\ &= 1 - \prod_{i=0}^{s-1} \left(1 - \frac{q}{4k^2 - i}\right) && (4.5) \end{aligned}$$

which can be re-written as Equation (4.1) by setting $q = (k+1)^2$. $\square$

Figure 4.6 shows how this probability varies with s samples for $k = 32$ and $k = 256$; each light client samples at least one unavailable share with about 60% probability after 3 samplings (*i.e.,* after querying respectively 0.07% of the block shares for $k = 32$ and 0.001% of the block shares for $k = 256$), and with more than

99% probability after 15 samplings (*i.e.,* after querying respectively 0.4% of the block shares for $k = 16$ and 0.005% of the block shares for $k = 256$). Figure 4.7 shows that light clients would have to download about 3.6 KB of shares to be able to detect incomplete blocks with more than 99% probability for $k = 32$, and about 57 bytes of shares for $k = 256$, for 1MB blocks.

Equation (4.6) shows a noticeable result: the probability $p_1(X \geq 1)$ is almost independent of k for large values of k; it is therefore convenient to have a large matrix size (*i.e.,* $k \geq 128$) as this reduces the amount of data that light clients have to download.

$$\lim_{k\to\infty} p_1(X \geq 1) = \lim_{k\to\infty} \left(1 - \prod_{i=0}^{s-1} \left(1 - \frac{(k+1)^2}{4k^2 - i}\right)\right) = 1 - (3/4)^s \tag{4.6}$$

Under the enhanced model described in Section 4.5.6, a malicious block producer could statistically link light clients based on the shares they query; *i.e.,* assuming that a light client would never request twice the same share, a block producer can deduce that any request for the same share comes from a different client. To mitigate this problem, light clients could sample without replacement by performing the procedure for sampling with replacement multiple times, and only stop when they have sampled s unique values.

Multi-Client Unrecoverable Block Detection

Theorem 9 captures the probability that more than $\hat{c}$ out of c light clients sample at least one unavailable share in a matrix with the minimum unavailable shares for unrecoverability.

Theorem 9. *Given a $2k \times 2k$ extended matrix E as shown in Figure 4.4, where $(k+1)^2$ shares are unavailable. If c players randomly sample $0 < s \leq (2k)^2$ shares from E, the probability that more than $\hat{c}$ players sample at least one unavailable share is:*

$$p_c(Y > \hat{c}) = 1 - \sum_{j=1}^{\hat{c}} \binom{c}{j} \left(p_1(X \geq 1)\right)^j \left(1 - p_1(X \geq 1)\right)^{c-j} \tag{4.7}$$

where $p_1(X \geq 1)$ is given by Equation (4.1).

Proof. We start by computing the probability that exactly $\hat{c}$ players sample at least one unavailable share; this probability is given by the binomial probability mass function:

$$p_{s,\hat{c}}(Y = \hat{c}) = \binom{c}{\hat{c}} \left(p_1(X \geq 1)\right)^{\hat{c}} \left(1 - p_1(X \geq 1)\right)^{c-\hat{c}} \tag{4.8}$$

where $p_1(X \geq 1)$ is given by Equation (4.1). Equation (4.8) describes the probability that $\hat{c}$ players succeed to sample at least one unavailable share. This can be viewed as the probability of observing $\hat{c}$ successes each happening with probability p_1, and $(c - \hat{c})$ failures each happening with probability $1 - p_1$; there are $\binom{c}{\hat{c}}$ possible ways of sequencing these successes and failures.

Equation (4.8) easily generalises to the binomial cumulative distribution function expressed in Equation (4.9)—the probability of observing at most $\hat{c}$ successes is the sum of the probabilities of observing j successes for $j = 1, \ldots, \hat{c}$.

$$p_c(Y \leq \hat{c}) = \sum_{j=1}^{\hat{c}} \binom{c}{j} \left(p_1(X \geq 1)\right)^{j} \left(1 - p_1(X \geq 1)\right)^{c-j} \tag{4.9}$$

Therefore the probability of observing more than $\hat{c}$ successes is given by Equation (4.10) below, which expands as Equation (4.7).

$$p_c(Y > \hat{c}) = 1 - p_c(Y \leq \hat{c}) \tag{4.10}$$

□

Figure 4.8 shows the variation of the number of light clients $\hat{c}$ for which $p_c(Y > \hat{c}) \geq 0.99$ with the sampling size s. The total number of clients is fixed to $c = 1000$, and the matrix sizes are $k = 64, 128, 256$; Equation (4.7) is however almost independent of k, as indicated by Equation (4.6). This figure can be used to determine the number of light clients that will detect incomplete matrices with high probability ($p_c(Y > \hat{c}) \geq 0.99$), and that there is little gain in increasing s over 15.

Recovery and Selective Share Disclosure

Corollary 1 presents the probability that light clients collectively samples enough shares to recover every share of the $2k \times 2k$ matrix.

If the light clients collectively sample all but $(k+1)^2$ distinct shares, the block producer cannot release any more shares without allowing the network to recover

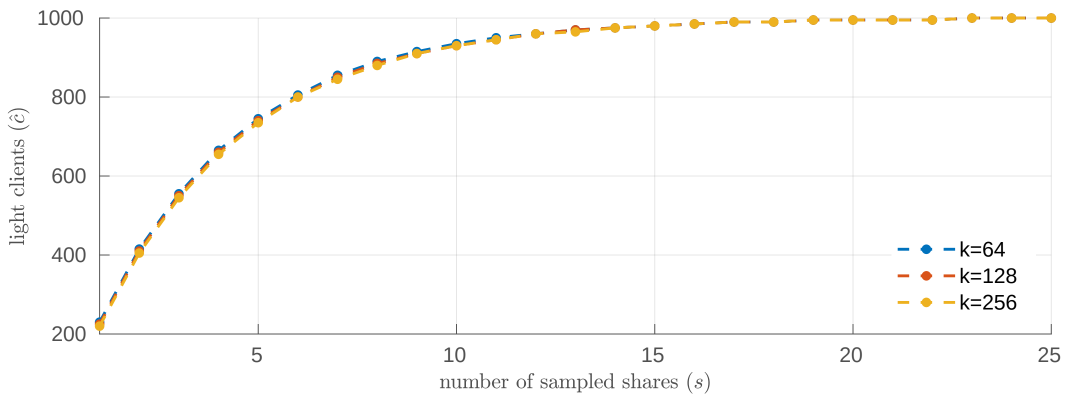


Figure 4.8: Plot of Equation (4.7)—variation of the number of light clients $\hat{c}$ for which $p_c(Y > \hat{c}) \geq 0.99$ with the sampling size s. The total number of clients is fixed to $c = 1000$, and the matrix sizes are $k = 64, 128, 256$; Equation (4.7) is however almost independent of k, as indicated by Equation (4.6).

the whole matrix; it follows from Theorem 7 that light clients need to collect at least:

$$\gamma = (2k)^2 - (k+1)^2 + 1 = k(3k-2)$$

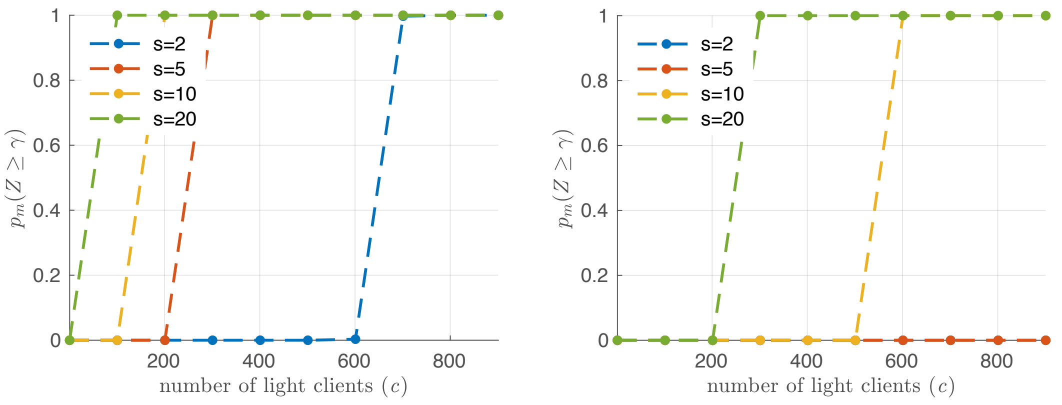
distinct shares (randomly chosen) to have the certainty to be able to recover the $2k \times 2k$ matrix. We are therefore interested in the probability that light clients—each sampling s distinct shares—collectively samples at least γ distinct shares; this probability is expressed by Corollary 1.

Theorem 10. *(Euler [61]) The probability that the number of distinct elements sampled from a set of n elements, after c drawings with replacement of s distinct elements each, is at least all but λ elements*[3]*:*

$$\begin{aligned} p_e(Z \geq n - \lambda) \quad &= \quad 1 - \sum_{i=1}^{\infty} (-1)^i \binom{\lambda + i - 1}{\lambda} \binom{n}{\lambda + i} (W_i)^c \qquad (4.11) \\ \textit{where} \qquad & W_i = \binom{n - \lambda - i}{s} \Big/ \binom{n}{s} \end{aligned}$$

Corollary 1. *Given a $2k \times 2k$ extended matrix E as shown in Figure 4.4, where each of c players randomly samples s distinct shares from E. The probability that the players collectively sample at least $\gamma = k(3k-2)$ distinct shares is $p_e(Z \geq \gamma)$*

[3]This problem is also known as *the coupon collector's problem* with group drawing [65].



(a) Matrix size $k = 16$ (b) Matrix size $k = 32$

Figure 4.9: Plot of Corollary 1—variation of the probability $p_e(Z \geq \gamma)$ with the number of clients (c) for different values of s and k.

Proof. Corollary 1 can be easily proven by substituting $\lambda = n - \gamma$ and $n = (2k)^2$ into Theorem 10. □

Figure 4.9 shows how $p_e(Z \geq \gamma)$ varies for different numbers of light clients, for matrix sizes $k = 16$ and $k = 32$. Contrarily to Equation (4.7), Figure 4.9 shows that $p_e(Z \geq \gamma)$ depends on the matrix size k. However, we can see that there is a small range in which $p_e(Z \geq \gamma)$ rises very quickly to 1.

Table 4.1 shows the minimum number of light clients (c) required to achieve $p_e(Z \geq \gamma) > 0.99$ for various values of k and s. As expected, a high number of samples s per clients reduces the number of clients needed to sample all the necessary shares. However, increasing k increases the total number of shares to sample, and thus increases the number of clients needed to sample all the necessary shares.

4.5.9 Properties Security Analysis

Standard Model

Corollary 2. *Under the standard model, a block producer cannot cause soundness (Definition 1) and agreement (Definition 2) to fail for more than c honest clients with a probability lower than* $p_1(X \geq 1)$ *per client, where c is determined by the probability distribution* $p_e(Z \geq \gamma)$.

Proof. Corollary 1 shows that with probability $p_e(Z \geq \gamma)$, c honest clients will sam-

$p_e(Z \geq \gamma)$	$s = 5$	$s = 10$	$s = 20$	$s = 50$
$k = 16$	277	138	69	28
$k = 32$	1,122	561	280	112
$k = 64$	4,516	2,258	1,129	451
$k = 128$	~18,000	~9,000	~4,500	1,811

Table 4.1: Minimum number of light clients (c) required to achieve $p_e(Z \geq \gamma) > 0.99$ for various values of k and s. The approximate values have been approached numerically as evaluating Theorem 10 can be extremely resource-intensive for large values of k.

ple enough shares to collectively recover the full block. Honest clients will gossip these shares to full nodes which then gossip them to each other, and within $k \times \delta$ at least one honest full node will then recover the full block data, thus satisfying soundness with a probability of $1 - p_1(X \geq 1)$ per client (the probability of the block producer not passing the client's random sampling challenge when all the block data is available).

If the data is available and no fraud proofs of incorrectly generated extended data was received by the client, then no other client will receive a fraud proof either, due to our assumption that there is at least one honest full node in the network and honest light clients are not under an eclipse attack, thus satisfying agreement with a probability of $1 - p_1(X \geq 1)$ per client.

Due to the selective share disclosure attack described in Section 4.5.6, this means that the block producer can violate soundness and agreement of the first c clients that make sample requests, as the block producer can stop releasing shares just before it is about to release the final shares to allow the block to be recoverable. □

Enhanced Model

Corollary 3. *Under the enhanced model, a block producer cannot cause soundness (Definition 1) and agreement (Definition 2) to fail with a probability lower than* $p_x(X \geq 1)$ *per client,*

$$p_x(X \geq 1) = \sum_{i=1}^{d} \frac{\binom{s}{i}\binom{s(c-1)}{d-i}}{\binom{c \cdot s}{d}} \tag{4.12}$$

where c is the number of clients and d is the number of requests that the block producer must deny to prevent full nodes from recovering the data.

Proof. The proof of Corollary 3 starts as the proof of Corollary 2; honest light clients collectively sample enough shares to recover the full block data by gossiping these shares to full nodes; soundness is satisfied with probability $1 - p_1(X \geq 1)$ per client. None of the light clients receive fraud proofs if the full data is available and no valid fraud proofs are sent over the network, and all light clients eventually receive a valid fraud proof if one is sent, satisfying agreement with the same probability.

However, the enhanced model assumes that all sample requests come through a perfect mix network (*i.e.,* requests are unlinkable between each other), and defeats the selective shares disclosure attack presented in Section 4.5.6. The enhanced model removes the notion of 'first' clients described in Corollary 2 as block producers cannot distinguish which requests comes from which client (since requests are unlinkable). Furthermore, if block producers randomly deny some requests, light clients would uniformly see some of their sample requests denied, and each light client would therefore consider the block invalid with equal probability.

Particularly, if c light clients each sample $0 < s \leq (2k)^2$ shares, block producers observe a total of $(c \cdot s)$ indistinguishable requests. Let us assume that a malicious block producer must deny at least d request to prevent full nodes from recovering the block data. The probability that a light client observes at least one of its requests denied (and thus rejects the block) is given by $p_x(X \geq 1)$ in Equation (4.12). The numerator of Equation (4.12) computes the number of ways of picking i of the denied requests among the s requests sent by the light client (i.e., $\binom{s}{i}$), multiplied by the number of ways to pick the remaining $d - i$ requests among the set of requests sent by other light clients: $c \cdot s - s = s(c-1)$ (i.e., $\binom{s(c-1)}{d-i}$). The denominator computes the total number of ways to pick any d requests out of the total number of requests (i.e., $\binom{c \cdot s}{d}$). The probability that at least one of the denied requests comes from a particular client is the sum of the probabilities for $i = 1, \ldots, d$. $\square$

Like Equation (4.1), Equation (4.12) rapidly grows and shows that light clients reject the block if invalid (for appropriate values of d). The value of d can be approximated using Corollary 1, and depends on s and c. To provide a quick intuition, if we assume that the light clients collectively sample at least once every share of the block, a malicious block producer must deny at least $(k+1)^2$ requests on different shares to prevent full nodes from recovering the block data; since multiple requests can sample the same shares, $d \geq (k+1)^2$.

4.6 Performance and Implementation

We implemented the data availability proof scheme described in Section 4.5 and a prototype of the state transition fraud proof scheme described in Section 4.4 in approximately 2.5k lines of Go code and released the code as a series of free and open-source libraries.[4]

Our Sparse Merkle tree implementation makes use of optimisation 2 as described in Section 2.1.2, thus the number of hash operations required per update is $O(s)$ where s is the number of non-default values in the tree.

For our Reed-Solomon coding implementation, we employ a coding algorithm based on Fast Fourier Transforms (FFT) [97, 127], and therefore the computational complexity for each encoding and decoding operation is $O(k \log(k))$ where k is the number of shares in the code.

We first evaluate the space and time complexity of the scheme in Section 4.6.1 and then present the performance benchmarks of our implementation in Sec-

[4]2D Reed-Solomon Merkle tree data availability scheme: `https://github.com/lazyledger/rsmt2d`

State transition fraud proofs prototype: `https://github.com/musalbas/fraudproofs-prototype`

Sparse Merkle tree library: `https://github.com/lazyledger/smt`

Object	**Worst case space complexity**
State fraud proof	$O(p+p\log(d)+w\log(s)+w)$
Availability fraud proof	$O(d^{0.5}+d^{0.5}\log(d^{0.5}))$
Single sample response	$O(\mathsf{shareSize}+\log(d))$
Block header with axis roots	$O(d^{0.5})$
Block header excl. axis roots	$O(1)$

Table 4.2: Worst case space complexity for various objects. p represents the number of transactions in a period, w represents the number of state witnesses for those transactions, d is short for $\mathsf{dataLength}$, and s is the number of key-value pairs in the state tree.

tion 4.6.2. We perform the measurements on a laptop with an Intel Core i5 1.3GHz processor and 16GB of RAM, and use SHA-256 for hashing.

4.6.1 Space and Time Complexity

Table 4.2 shows the space complexity for different objects. We observe that the size of the state transition fraud proofs only grows logarithmically with the size of the block and state, whereas the availability fraud proofs (as well as block headers with the axis roots) grows at least in proportion to the square root of the size of the block, due to our two-dimensional erasure coding.

Table 4.3 shows the time complexity for various actions. For generating and verifying fraud proofs, the primary cost is processing the state transitions, as well as generating and verifying the Merkle proofs for transactions and witnesses. However, verifying fraud proofs is less expensive, as only the state transitions for the (few) transactions in a single period need to be processed, rather than the whole block. Similarly, verifying a data availability fraud proof is cheaper than generating one, because only a single coding operation with $O(d^{0.5}\log(d^{0.5}))$ complexity for one row or column needs to performed, rather than for every row and column requiring $O(d\log(d^{0.5}))$ complexity.

Action	Worst case time complexity
[G] State fraud proof	$O(b + p\log(d) + w\log(s))$
[V] State fraud proof	$O(p + p\log(d) + w)$
[G] Availability fraud proof	$O(d\log(d^{0.5}) + d^{0.5}\log(d^{0.5}))$
[V] Availability fraud proof	$O(d^{0.5}\log(d^{0.5}))$
[G] Single sample response	$O(\log(d^{0.5}))$
[V] Single sample response	$O(\log(d^{0.5}))$

Table 4.3: Worst case time complexity for various actions, where [G] means generate and [V] means verify. p represents the number of transactions in a period, b represents the number of transactions in the block, w represents the number of state witnesses for those transactions, d is short for `dataLength`, and s is the number of key-value pairs in the state tree. For generating and verifying state fraud proofs, we assume that each transaction takes the same amount of time to process. For generating fraud proofs, we also include the cost of verifying the block itself.

4.6.2 Benchmarks

Table 4.4 shows the size of various objects when transmitted over the network. We observe that the size of the block only causes a marginal impact on the size of the state fraud proof (a logarithmic increase, as discussed in Section 4.6.1); this is because the number of transactions in a period remains static, but the size of the Merkle proof for each transaction increases slightly. Block size impacts the size of availability fraud proofs and the axis roots more, as the size of a single row or column is proportional to the square root of the size of the block.

Table 4.5 shows the computation time for generating and verifying various objects; the benchmark for state fraud proof generation includes time spent verifying the block. As expected and discussed, verifying state transition fraud proofs and availability fraud proofs is significantly quicker than generating them.

Comparison with Bitcoin. We provide a comparison of the bandwidth (space complexity) costs to sync the chain if Bitcoin adopts light clients with data avail-

Object (10 tx/period)	**Size ($\sim$0.25MB block)**	**Size ($\sim$1MB block)**
State fraud proof	14,090b	14,410b
Availability fraud proof	12,320b	26,688b
Single sample response	320b	368b
Block header with. axis roots	2,176b	4,224b
Block header excl. axis roots	128b	128b

Table 4.4: Illustrative sizes for objects for $\sim$0.25MB and $\sim$1MB blocks, assuming that a period consists of 10 transactions, the average transaction size is 225 bytes, and that conservatively there are 2^{30} non-default nodes in the state tree.

ability proofs support, using historical block data. As of June 2020, 268GB of data needs to be downloaded to bootstrap a full node, and 48MB of data needs to be downloaded to bootstrap a light client. In order to bootstrap a fraud proof enabled light client that verifies data availability proofs, the client would have to download 6GB of data, assuming 256-byte shares and 15 samples per block. This is 2.2% of the data that a full node needs to download.

4.7 Conclusion

We presented, implemented and evaluated a complete fraud and data availability proof scheme, which enables light clients to have security guarantees almost at the level of a full node, with the added assumptions that there is at least one honest full node in the network that distributes fraud proofs within a maximum network delay, and that there is a minimum number of light clients in the network to collectively recover blocks.

Action	**Time (∼0.25MB block)**	**Time (∼1MB block)**
[G] State fraud proof	41.22 ms	182.80 ms
[V] State fraud proof	0.03 ms	0.03 ms
[G] Availability fraud proof	4.91ms	19.18ms
[V] Availability fraud proof	0.05ms	0.08ms
[G] Single sample response	< 0.00001ms	< 0.00001ms
[V] Single sample response	< 0.00001ms	< 0.00001ms

Table 4.5: Computation time (mean over ten repeats) for various actions, where [G] means generate and [V] means verify. We assume that a period consists of 10 transactions, the average transaction size is 225 bytes, and each transaction writes to one key in the state tree.

Chapter 5

LazyLedger: A Distributed Data Availability Ledger With Client-Side Smart Contracts

Mathematics is being lazy. Mathematics is letting the principles do the work for you so that you do not have to do the work for yourself.

George Pólya

5.1 Introduction

So far, blockchain-based distributed ledger platforms such as Bitcoin [109] and Ethereum [36] have adopted similar consensus design paradigms, where the validity of the blocks proposed by block producers is determined by (*i*) whether it is the block producer's turn to propose a block (state agreement) and (*ii*) whether the transactions in the block are valid according to some state machine (state validity). Traditional consensus protocols such as Practical Byzantine Fault Tolerance [43] have also taken a similar approach, where consensus nodes (replicas) process transactions according to a state machine.

The scalability issues that have plagued decentralised blockchains [49] can be attributed to the fact that in order to run a node that validates the blockchain, the node must download, process and validate every transaction included in the chain. As a result, various scalability efforts have emerged including on-chain scaling via sharding (Chapter 3), which aims to split the state of the blockchain into multiple shards so that transactions can be processed by different consensus groups in parallel. On the other hand, off-chain scaling via state channels [123, 108] takes the approach of moving transactions off-chain and using the blockchain as a settlement layer.

However, it is also worth exploring alternative blockchain design paradigms that may be suitable for different types of applications, where nodes that validate the blockchain do not need to validate the contents of the blocks. Instead, the end-users of applications that store information on the blockchain can be concerned with the validation of such contents. This would remove the bottleneck where nodes need to validate everyone else's transactions, and reducing the problem of validating the blockchain to simply verifying that the contents of the block are available (the data availability problem), so that end-users can meaningfully access the information needed to apply state transitions on their applications. In such a paradigm, the blockchain is used solely for ordering and making available messages, rather than executing and verifying the state machine transitions of transactions. Because messages for applications are executed by end-users off-chain, the logic of these applications does not need to be defined on-chain, and thus application logic can be written in any programming language or environment, and changing the logic does not require a hard-fork of the chain.

A result of reducing blockchain validation to the data availability problem is that one can fully achieve consensus on new messages without downloading the entire set of messages, using probabilistic data availability verification techniques (Section 4.5), as consensus participants do not need to process the contents of messages.

Philosophically, LazyLedger can be thought of as a system of 'virtual' sidechains [19] that live on the same chain, in the sense that transactions associated with each application only need to be processed by users of those applications, similar to the fact that only users of a specific sidechain need to process transactions of that sidechain. However, because all applications in LazyLedger share the same chain, the availability of the data of all their transactions are equally and uniformly guaranteed by the same consensus group, unlike in traditional sidechains where each sidechain may have a different (smaller) consensus group.

In this chapter, we make the following contributions:

- We design a blockchain, LazyLedger, where consensus and transaction validity is decoupled, and compare two alternative block validity rules which just ensure that block data is available. One is a simple rule where nodes simply download the blocks themselves, and the other uses data availability proofs as described in Section 4.5, which is probabilistic but more efficient as nodes do not need to download entire blocks.

- We build an application-layer on top of our proposed blockchain, where end-user clients can efficiently query the network for data relating only to their applications, and only need to execute transactions related to their applications.

- We implement and evaluate several example LazyLedger applications; including a currency, a name registrar and a dummy application for testing purposes.

5.2 Related Work

5.2.1 Mastercoin

Mastercoin (now OmniLayer) [151] is a blockchain application system predating Ethereum [36], which uses Bitcoin as a protocol layer for posting messages. This is similar to LazyLedger in the sense that the blockchain can be used to post arbitrary messages that are interpreted by clients, however in Mastercoin all nodes must download all Mastercoin messages as the Bitcoin base layer does not support efficient data availability schemes using probabilistic data availability proofs.

Additionally, Mastercoin has a set of hardcoded applications, and does not support arbitrary applications. In contrast, LazyLedger examines what an ideal new blockchain would look like for use as a base layer in a system where the base layer is only for posting messages and data availability.

5.2.2 Separating Agreement from Execution for BFT Services

Yin *et al.* [155] have proposed an architecture for BFT state machine replication (in a non-blockchain setting) where message ordering is separated from execution—a key idea of LazyLedger. In their system, the agreement layer is responsible for receiving and ordering client requests and sending them in order to the execution layer. The execution layer then executes the request, sending the reply back to the agreement layer in the form of a 'reply certificate', which relays it to the client that invoked the operation.

This reduces state machine replication costs as only $2f+1$ execution replicas are needed to tolerate f faults (a simple majority), whereas $3f+1$ agreement replicas are needed. This is in contrast to BFT architectures where agreement and execution are combined, where $3f+1$ replicas are needed.

LazyLedger differs from this architecture in several ways. Firstly, as LazyLedger operates in a decentralised setting, public verifiability of transactions is important. Therefore the agreement layer in LazyLedger (the base chain) ought to be responsible not only for ordering but data availability, and thus it is also a data availability layer. This is because the availability of block data need to be guaranteed in order for relevant transactions to be validated and processed by public nodes.

Secondly, in LazyLedger there is no explicit notion of execution replicas that communicate with the agreement layer, and abstracts the separation further. LazyLedger applications (which can be thought of as execution layers) post messages directly on the LazyLedger base chain, which are interpreted locally by users of that application who watch the LazyLedger chain for messages for their application. Therefore the LazyLedger base chain does not need to be aware of, or understand, any applications building on top of it, and 'reply certificates' are not generated.

5.2.3 Hyperledger

Hyperledger Fabric [42, 140] is a permissioned blockchain project that also adopts an architecture for separating agreement from execution. There is an 'ordering service' which orders transactions received by 'endorsing peers', which execute and endorse transactions received by clients. In comparison to the architecture proposed by Yin *et al.* [155], in Hyperledger Fabric it is transactions endorsed by the execution layer that are ordered, rather than client requests that are to be sent to the execution layer.

Compared to LazyLedger, this architecture does not fully decouple agreement from execution, as requiring the ordering service to check that every transaction has the endorsement of endorsing peers, requires an execution layer that performs an operation on each transaction. A key design goal of LazyLedger is to enable nodes validating the base chain to be able to validate blocks without needing to read the contents of, or perform operations on all transactions in blocks, thus truly separating agreement from execution.

5.2.4 Sidechains and Optimistic Rollups

Sidechains [19, 122, 1] are blockchains that can inherit and perform operations on the state or assets of some parent 'main' chain. They have been proposed as a layer two scalability solution, as transactions are processed off the main chain.

In traditional sidechain designs [19, 122], a sidechain has its own consensus group that produces sidechain blocks. Users can transfer state or assets to the sidechain, perform transactions on the sidechain, and then transfer the state or assets back to the main chain. This is effectively a two-way asset bridge between the main chain and the sidechain.

In the Plasma sidechain design [122], a user can enter a sidechain by sending assets to a smart contract. The validators in the sidechain follow the transactions in the main chain, and can then observe and recognise that new assets have entered the sidechain. The user then performs transactions within the sidechain. To exit the sidechain, the user can present a light client Merkle proof to the smart contract on the main chain showing proof that they own assets on the sidechain, using the sidechain's state root (see Section 2.2.5). The smart contract then releases the assets back to the user on the main chain. The pegged Bitcoin sidechain design also follows a similar architecture using SPV proofs [19].

In this sidechain architecture, the user trusts the consensus in the sidechain to only produce blocks that contain valid state. Malicious consensus nodes in a sidechain can produce blocks with invalid state that steals assets in the sidechain by incorrectly changing their ownership. The malicious party can then submit Merkle proofs to the main chain smart contract to show that they own the assets, and thus steal the assets.

Plasma attempts to solve this by allowing users to submit fraud proofs to the main chain smart contract in case the sidechain produces blocks with invalid state. However as discussed in Chapter 4, the generation of fraud proofs requires the sidechain block data to be available. In order to deal with cases where sidechain block data is withheld, Plasma proposes a 'mass-exit' protocol where users can collectively force the entire state of the sidechain to be posted and processed on the main chain. This has several drawbacks, including the fact that it is not practical to congest the main chain by posting the state of an entire sidechain within a short period of time, and users need to constantly watch the chain in case they need to participate in a mass-exit [125].

Optimistic rollups [1, 38] is a sidechain design that works around the data availability problem in Plasma, by simply requiring sidechain blocks to be posted on the main chain. Assuming that sidechain clients validate the data availability of the main chain blocks, then the data of sidechain blocks is guaranteed to be available, and thus fraud proofs can be generated. This avoids the need for a mass-exit protocol.

In this architecture, the sidechain has on-chain (main chain) data availability but off-chain execution, as the consensus nodes and validators on the main chain do not execute any of the transactions in the sidechain, but simply guarantee their availability. A benefit of this scheme is that as blocks are posted on the main chain, the main chain is responsible for ordering and data availability of sidechain blocks. Therefore the sidechain inherits the security of the consensus of the main chain. This means that an optimistic rollup chain can be run by a single block producer known as an ‘aggregator’ (which collects transactions and ’rolls’ them up into blocks), as this block producer does not need to be trusted neither for ordering or state validity. If the aggregator produces rolls up blocks with invalid state, then a fraud proof will be generated so that blocks can be rejected by the sidechain users or the on-chain smart contract in the case of a two-way bridge. Additionally, if the aggregator loses liveness, then a new aggregator can take over via an aggregator selection mechanism [1].

LazyLedger and optimistic rollup sidechains are complementary, as optimistic rollups can be implemented as a LazyLedger applications that uses LazyLedger as an efficient data availability layer.

5.3 Model

5.3.1 Threat Model and Nodes

We consider the following types of nodes in LazyLedger:

- **Consensus nodes.** These are nodes which participate in the consensus set, to decide which blocks should be added to the chain.
- **Storage nodes.** These are nodes which store a copy of all of the data in the blockchain and its blocks.
- **Client nodes.** These are effectively the end-users of the blockchain system. They participate in applications or use cases that use the blockchain, and receive transaction data or messages from storage nodes relevant to their applications.

LazyLedger makes use of data availability proofs as described in Section 4.5, and therefore inherits the network and threat model in Section 4.3.3. We will resummarise the model here.

We assume a synchronous gossiping network [34] amongst these nodes. We assume that there is a maximum network delay δ so that if an honest node can receive a message from the network at time T, then any other honest node can also do so at time $T' \leq T + \delta$.

In LazyLedger all node types may have some connections with any other node type and thus the topology of the network is non-hierarchical. However, client nodes may wish to ensure that they are connected to at least one storage node if they wish to utilise their services. We assume that the graph of honest nodes is well connected, as this is necessary to achieve a synchronous gossiping network.

For data availability proofs, we assume a minimum number of honest nodes (of any type) in the network to allow for a block to be reconstructed, as each node downloads a small chunk of every block. The specific number depends on the parameters of the system, and is analysed in Section 4.5.8. In the LazyLedger setup, as there is no notion of a 'fully-validating node' for clients to gossip shares to, block shares are instead gossiped to storage nodes which are responsible for reconstructing blocks if block data is missing.

5.3.2 Block Model

Similar to Chapter 4, we assume a blockchain data structure that at minimum consists of a hash-based chain of block headers $H = (h_0, h_1, ...)$. Each block header h_i contains the root mRoot_i of a Merkle tree of a list of messages $M_i = (m_i^0, m_i^1, ...)$, such that given a function $\mathsf{root}(M)$ that returns the Merkle root of a list of messages M, then $\mathsf{root}(M_i) = \mathsf{mRoot}_i$. This is not an ordinary Merkle tree, but an ordered Merkle tree we refer to as a 'namespaced' Merkle tree which we describe in Section 5.5.3. A block header h_i is considered to be valid if given some boolean function

$$\mathsf{blockValid}(h) \in \{\mathsf{true}, \mathsf{false}\}$$

then $\mathsf{blockValid}(h_i)$ must return true.

If a block is valid, then it has the potential to be included in the blockchain. We assume that the blockchain has some consensus rules to decide which valid blocks have consensus to be included in the blockchain, and resolve forks. A block header

h_i is considered to have consensus if given some boolean function

$$\mathsf{inChain}(h, \{H_0, H_1, ...\}) \in \{\mathsf{true}, \mathsf{false}\}$$

then $\mathsf{inChain}(h_i, \{H_0, H_1, ...\})$ must return true, where each H_j is a distinct chain of block headers and $\{H_0, H_1, ...\}$ is the set of distinct chains observed (there may be multiple in the event of a fork).

Note that computing $\mathsf{inChain}$ on h_i can only return true if and only if $\mathsf{blockValid}(h_i)$ returns true, regardless of the forks to pick from. Apart from this constraint, the specific consensus rules used by $\mathsf{inChain}$ are arbitrary and are out of scope for the design of LazyLedger. For example, $\mathsf{inChain}$ may use proof-of-work or proof-of-stake with the longest chain rule [109, 20].

5.3.3 Goals

With this threat model in mind, LazyLedger has the following goals:

In the text below, 'messages relevant to the application' means messages that are necessary to compute the state of an application, and is discussed in more depth in Section 5.5.2.

1. **Availability-only block validity.** The result of $\mathsf{blockValid}(h_i)$ should be true if the data behind mRoot_i is available to the network. This consequently means that consensus nodes should not need to execute or verify messages in blocks.

2. **Application message retrieval partitioning.** Client nodes must be able to download all of the messages relevant to the applications they use from storage nodes, without needing to downloading any messages for other applications.

3. **Application message retrieval completeness.** When client nodes download messages relevant to the applications they use from storage nodes, they must be able to verify that the messages they received are the complete set of messages relevant to their applications, for specific blocks, and that there are no omitted messages.

4. **Application state sovereignty.** Client nodes must be able to execute all of the messages relevant to the applications and compute the state for their applications, without needing to read the state of other applications, unless other specific applications are explicitly declared as dependencies.

5.4 Block Validity Rule Design

The key idea of LazyLedger is that the result of $\mathsf{blockValid}(h_i)$ should only depend on whether the data required to compute mRoot_i is available to the network or not, rather on whether any of the messages in the block correspond to transactions that satisfy the rules of some state machine (Goal 1 in Section 6.3.3). This way, we can decouple transaction validity rules from the consensus rules, as the result of $\mathsf{inChain}$ does not depend on the contents of the messages in the block M_i, when $\mathsf{blockValid}(h_i)$ is computed (recall $\mathsf{inChain}$ on h_i can only return true if and only if $\mathsf{blockValid}(h_i)$ returns true).

We consider that checking the availability of the data necessary to recompute mRoot_i is the bare minimum necessary requirement to have a useful functioning blockchain. This is because, as we shall see in Section 5.5, clients need to know the transactions that have occurred in the blockchain in order to know the state of applications on the blockchain and thus do anything useful. If the data behind a block is unavailable, clients would not be able to compute the state of their applications.

We compare two possible validity rules with different trade-offs. Section 5.4.1 describes a simple validity rule that satisfies Definition 1 and Definition 2 (defined in Section 4.5) with 100% probability, for an $O(n)$ bandwidth cost where n is the size of the block, because the node must download the entire block data to confirm that it is available. Section 5.4.2 is the probabilistic validity rule that uses data availability proofs (described in Section 4.5) that satisfies Definition 1 and Definition 2 with a high but less than 100% probability, but with a $O(\sqrt{n}+\log(\sqrt{n}))$ bandwidth cost because the block's row and column Merkle roots and only a static number of samples and their logarithmically-sized Merkle proofs from the block need to be downloaded. This bandwidth cost is analysed further in Section 5.6.

5.4.1 Simplistic Validity Rule

In the Simplistic Validity Rule, $\mathsf{blockValid}(h_i)$ returns true if and only if upon receiving a block header h_i from the network, the node is also able to download M_i from the network and authenticate that the Merkle root of the downloaded M_i is mRoot_i, by checking that $\mathsf{root}(M_i) = \mathsf{mRoot}_i$.

Upon $\mathsf{blockValid}(h_i)$ returning true, the node must distribute h_i and M_i to the nodes it is connected to, should the nodes request the data if they do not have it. The node should thus store M_i for at least δ, the maximum network delay.

Theorem 11. *The Simplistic Validity Rule satisfies Definition 1 (Soundness).*

Proof. If $\mathsf{blockValid}(h_i)$ returns true on an honest node, then the node will distribute M_i to the nodes it is connected to, of at least one of which is honest, and will also run $\mathsf{blockValid}(h_i)$ and distribute M_i, and so on. Thus a storage node will receive M_i within the maximum network delay δ, which there exists at least one of which is honest. □

Theorem 12. *The Simplistic Validity Rule satisfies Definition 2 (Agreement).*

Proof. If $\mathsf{blockValid}(h_i)$ returns true on an honest node, then the node will distribute M_i to the nodes it is connected to, of at least one of which is honest, and will also run $\mathsf{blockValid}(h_i)$ and distribute M_i, and so on. Thus all honest nodes will receive M_i within the maximum network delay δ, and $\mathsf{blockValid}(h_i)$ will thus return true, causing them to accept h_i as an available block. □

5.4.2 Probabilistic Validity Rule

For the Probabilistic Validity Rule, $\mathsf{blockValid}(h_i)$ utilises the probabilistic data availability scheme based on random sampling the erasure coded version of the block data M_i described in Section 4.5. Unlike the Simplistic Validity Rule, this scheme is probabilistic in satisfying these definitions, however it is more efficient because only a part of the block data needs to be downloaded to obtain high probability guarantees that the data is available; see Section 4.5.8 for analysis.

The bandwidth cost of executing $\mathsf{blockValid}(h_i)$ is $O(\sqrt{n} + \log(\sqrt{n}))$ where n is the size of the block, because each node needs to download $2\sqrt{n}$ row and column Merkle roots for the block's 2D erasure coded data, and a fixed number of share samples and their corrosponding Merkle proofs authenticating them to one of the block's row or column roots (which are logarithmic in size). Further analysis specific to LazyLedger will be provided in Section 5.6.

As mentioned in Section 4.5.6, this scheme only works if there is a sufficient minimum number of nodes in the network making a sufficient number of sample requests so that the network collectively samples enough shares to be able to reconstruct the block, thus the maximum block size and number of samples each node makes should be set to reasonable values such that this condition is met.

Using the Probabilistic Validity Rule, in order to securely scale the block size of the chain, one can increase the number of nodes in the network, in order to ensure that the minimum number of nodes assumption requires by data availability proofs holds true. See Table 4.1 in Chapter 4 for example parameterisation and numbers for the minimum nodes that are required for certain block sizes.

5.5 Application-Layer Design

5.5.1 Application Model

Recall in Section 5.3 that LazyLedger has client nodes which read and write messages in blocks relevant to their application, and that the contents of blocks have no validity rules, and thus any arbitrary message can be included in a block. LazyLedger applications are akin to smart contracts, with the primary difference being that they are executed by end-user clients rather than consensus participants. Thus, application logic is defined and agreed upon entirely off-chain by clients of that application, and may therefore be written in any programming language or environment.

A client can submit a message to be recorded on the blockchain that specifies a transaction for a specific application, which will then be read and parsed by other clients of that applications, which may then modify their copy of the state of that application.

Applications are identified by their own 'namespace', and well-formed mes-

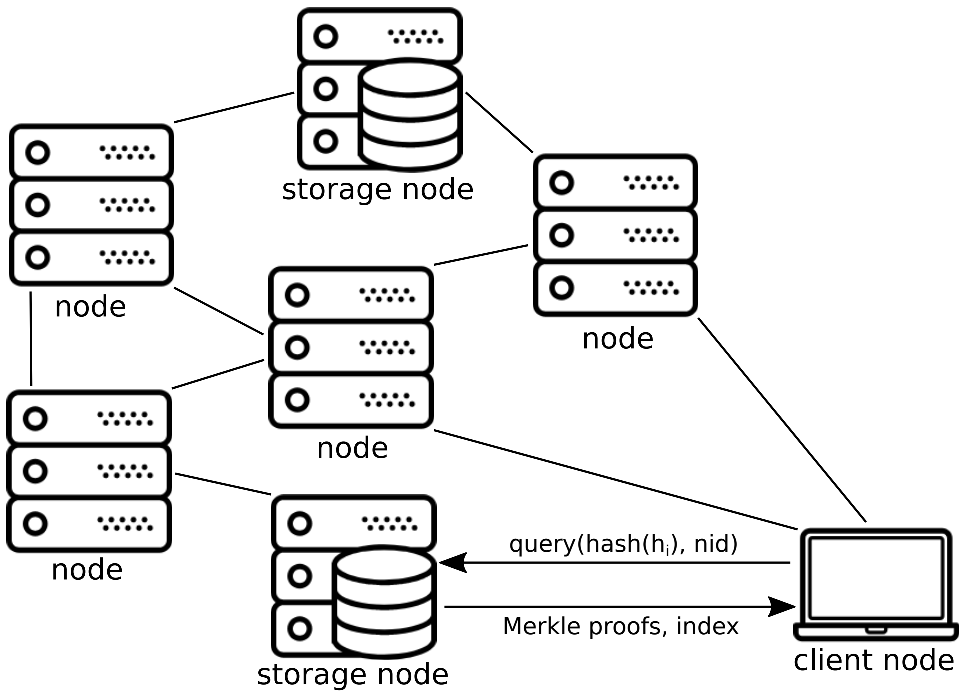


Figure 5.1: Overview of interaction between client nodes and storage nodes.

sages associated with a specific application can be parsed to determine their application namespace. We assume a function $\mathsf{ns}(m)$ that takes as input a message m and returns its namespace ID. Therefore if a client is a user of an application with ID nid, it is interested in reading all messages m in the ledger such that $\mathsf{ns}(m) = \mathsf{nid}$, in order to determine the state of its application.

Since the consensus of the blockchain does not require checking the validity of any transactions included in the blockchain, the ledger may include transactions that are considered invalid according to the logic of certain applications. Therefore we define a state transition function that LazyLedger applications should use that does not return an error. Given an application with ID nid:

$$\mathsf{transition}_{\mathsf{nid}}(\mathsf{state}, \mathsf{tx}) = \mathsf{state}'$$

$\mathsf{transition}_{\mathsf{nid}}(\mathsf{state}, \mathsf{tx})$ cannot return an error because if an adversarial actor includes an invalid tx in a block, then the state of the application would end up in a permanently erroneous state. Therefore if tx is considered erroneous by the logic of the transition function, it should simply return the previous state, state, as the new state.

Clients who use an application with ID nid should agree with each other on the logic or code of $\mathsf{transition}_{\mathsf{nid}}$. If one client decides to use different logic for $\mathsf{transition}_{\mathsf{nid}}$, then that client would reach a different final state for that application than everyone else, which in effect means that they would be using a different 'hard-fork' of the application, but it would not effect the state of anyone else.

Interestingly, this means that it is possible for users of an application to decide to change the logic of that application without requiring a hard-fork of the LazyLedger blockchain that would effect other applications. Instead, they would only be hard-forking their application. However if immutability of the logic is important, the creator of the application may decide for example that the namespace identifier of the application should be the cryptographic hash of the application's logic, as a convention.

5.5.2 Cross-Application Calls

Some applications may want to read or write to the state of other applications (*i.e.,* a cross-contract call) on LazyLedger. As LazyLedger does not have an on-chain execution environment, we do not enforce a particular cross-application call mechanism, and so this is out-of-scope and left to the developer. However, we discuss some of the options here. We consider two scenarios in which an application may want to do this: either as a read or a write.

Recall in Section 6.3.3 that Goal 4 of LazyLedger is application state sovereignty, which means that users of an application should not have to execute messages from other irrelevant applications. An application can specify other applications as dependencies in its logic, where knowledge of the state of the dependency applications is necessary in order to compute the state of the application. An application B is thus defined as 'relevant' to users of application A if B is a dependency of A, however if A is not a dependency of B, then A is not relevant to the users of B. In order to preserve the notion of state sovereignty, this means third party applications cannot force other applications to take a dependency on the state of third party applications.

Reads (One-Way Dependency)

In the case of a read, an application may have a function that can only be executed if another application that it depends on is in a certain state. In such a case, in order to validate that a pre-condition is met, clients of an application can also download and verify the state of the application's dependency applications; however the clients of the dependency application do not need to download the state of the applications which depend on it. Therefore application state sovereignty is preserved.

For example, consider a name registrar application where clients can register names only if they send money to a certain address in a different currency application. The clients of the name registrar application would have to also become clients of the currency application, in order to verify that when a name is registered, there is a corresponding transaction that sends the funds to pay for the name to the correct address.

In order to avoid the need for an application's clients and dependants to have to download and verify all the transactions of the application in order to read its state, an application can adopt the optimistic rollup sidechain model (Section 5.2.4), where the application has an 'aggregator' that periodically rolls the transactions up into block headers that contain state roots that represent the state. Instead of clients needing to download and verify all the transactions of an application in order to perform a read operation, clients can operate as light clients and download the latest state root and request inclusion proofs of state from nodes that have the full state.

Writes (Two-Way Dependency)

In the case of a write, an application A may want to modify the state of another application B during a transaction. If we assume a model where smart contracts cannot store their own secret state (as the blockchain is public) and thus own private keys, this is only possible if application B is designed so that application A can be a dependency of application B. This is because in order to execute the write, the clients of application B would have to download and verify the state of application A, to verify that it has the authority to execute the write (*i.e.,* certain pre-conditions are met), as the smart contract cannot own an account on B controlled by a private

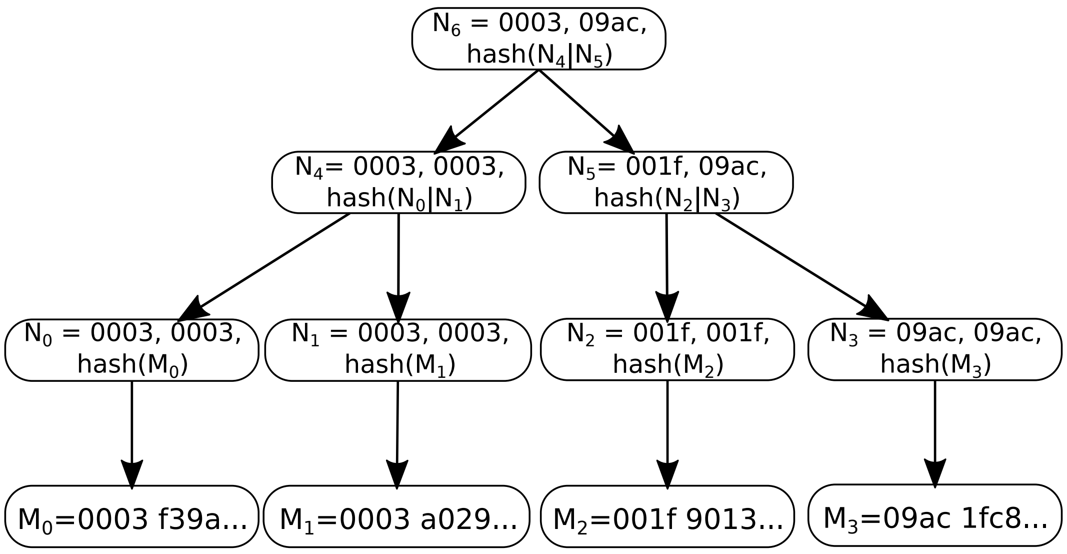

N6 = 0003, 09ac, hash(N4|N5)
N4= 0003, 0003, hash(N0|N1)
N5= 001f, 09ac, hash(N2|N3)
N0 = 0003, 0003, hash(M0)
N1 = 0003, 0003, hash(M1)
N2 = 001f, 001f, hash(M2)
N3 = 09ac, 09ac, hash(M3)
M0=0003 f39a...
M1=0003 a029...
M2=001f 9013...
M3=09ac 1fc8...

Figure 5.2: An example of a namespaced Merkle tree.

key. This therefore creates a two-way dependency, where both applications take a dependency on each other.

If any application was allowed to execute a write in any application, then it would mean that clients would have to download and verify other applications against their will, thus violating Goal 4 in Section 6.3.3 (application state sovereignty). Thus applications that support writes from other applications must explicitly be designed in a way that they allow other applications to add themselves as a dependency. This can be done for example by allowing an application *A* to post its state roots to application *B*, so that *B* can verify state inclusion proofs from *A*. This is similar to the design of Plasma [122] and optimistic rollups [1], where sidechains post their state roots to an Ethereum smart contract, so that Ethereum main chain is able to read state from the sidechains via state inclusion proofs.

5.5.3 Storage Nodes and Namespaced Merkle Tree

In order to satisfy Goal 2 in Section 6.3.3 (application message retrieval partitioning) to allow client nodes to be able to retrieve all the messages relevant to the application namespaces they are interested in without having to download and parse the entire blockchain themselves (*e.g.,* if they use the Probabilistic Validity Rule, or simply assume that the consensus has a honest-majority that only accepts available blocks), they may query storage nodes for all of the messages in a particular application namespace for particular blocks. The storage node can then return Merkle proofs the relevant messages being included in the blocks.

In order to allow storage node to prove to clients that they have returned the complete set of messages for a namespace included in a block's Merkle tree of messages (Goal 3 in Section 6.3.3, application message retrieval completeness), we use a 'namespaced' Merkle tree described below, which is an ordered Merkle tree that uses a modified hash function so that each node in the tree includes the range of namespaces of the messages in all of the descendants of each node. The leafs in the tree are ordered by the namespace identifiers of the messages.

In a namespaced Merkle tree, each non-leaf node in the tree contains the lowest and highest namespace identifiers found in all the leaf nodes that are descendants of the non-leaf node, in addition to the hash of the concatenation of the children of the node. This enables Merkle inclusion proofs to be created that prove to a verifier that all the elements of the tree for a specific namespace have been included in a Merkle inclusion proof.

This is inspired by the 'flagged' Merkle tree concept by Crosby and Wallach [50], where each node in the tree has a flag that represents the attributes that its leafs represents.

The Merkle tree can be implemented using standard unmodified Merkle tree algorithms, but with a modified hash algorithm that depends on an existing hash function, that prefixes hashes with namespace identifiers. Suppose $\mathsf{hash}(x)$ is a cryptographically secure hash function such as SHA-256. We define a wrapper function $\mathsf{nsHash}(x)$ that produces hashes prefixed with namespace identifiers. A namespaced hash has the format $\mathsf{minNs}||\mathsf{maxNs}||\mathsf{hash(x)}$, where minNs is the lowest namespace identifier found in all the children of the node that the hash represents, and maxNs is the highest.

The value of minNs and maxNs in the output of $\mathsf{nsHash}(x)$ depends on if the input x is a leaf or two concatenated tree nodes, as illustrated by Figure 5.2. If x is a leaf, then $\mathsf{minNs} = \mathsf{maxNs} = \mathsf{ns}(x)$, as the hash contains only one leaf with a single namespace.

If x is two concatenated tree nodes, then $x = \mathsf{left}||\mathsf{right}$ where $\mathsf{left} = \mathsf{leftMinNs}||\mathsf{leftMaxNs}||\mathsf{hash(x)}$ and $\mathsf{right} = \mathsf{rightMinNs}||\mathsf{rightMaxNs}||\mathsf{hash(x)}$. Thus in the output of $\mathsf{nsHash}(x)$, $\mathsf{minNs} = \mathsf{min}(\mathsf{leftMinNs}, \mathsf{rightMinNs})$ and $\mathsf{maxNs} = \mathsf{max}(\mathsf{leftMaxNs}, \mathsf{rightMaxNs})$.

An adversarial consensus node may attempt to produce a block that contains a Merkle tree with children that are not ordered correctly. To prevent this, we can set a condition in nsHash such that there is no valid hash when $\mathsf{leftMaxNs} \geq \mathsf{rightMinNs}$, and thus there would be no valid Merkle root for incorrectly ordered children. Therefore $\mathsf{blockValid}(h_i)$ would return false in the simplistic and probabilistic validity rules as there is no possible M_i where $\mathsf{root}(M_i) = \mathsf{mRoot}_i$. Additionally, recall that $\mathsf{root}(M_i) = \mathsf{mRoot}_i$ and thus $\mathsf{blockValid}(h_i)$ would also return false if the Merkle root of the tree is constructed incorrectly, *e.g.,* if the minimum and maximum namespaces for a node in the tree are incorrectly labelled.

Because only the hash function is being modified in the Merkle tree, the Merkle tree is generated, and Merkle proofs are verified using standard algorithms. However, during Merkle proof verification, an extra step is necessary in order to verify that the proofs cover all of the messages for a specific namespace.

A client node can send a query $\mathsf{query}(\mathsf{hash}(h_i), \mathsf{nid})$ to a storage node to request all of the messages in block h_i that have namespace ID nid. The storage node replies with a list of Merkle proofs $\mathsf{proofs} = (\mathsf{proof}_0, \mathsf{proof}_1, ..., \mathsf{proof}_n)$ and an index index that specifies the index in the tree in which proof_0 is located. In addition to the client node verifying all the proofs, the client node also verifies that the highest namespace in all of the left siblings included in proof_0 are smaller than nid, and the lowest namespace in all of the right siblings included in proof_n are larger than nid.

If a block has no messages associated with nid, then only one proof proof_0 is returned which corresponds to the child in the tree where the child to the left of it is smaller than nid but the child to the right of it is larger than nid. The actual message in the child does not need to be included in the proof as the purpose of the proof would just be to show that there are no messages in the tree for nid.

Theorem 13. *Assuming the Merkle tree is correctly constructed, an incomplete set of Merkle proofs* $\mathsf{proofs} = (\mathsf{proof}_0, \mathsf{proof}_1, ..., \mathsf{proof}_n)$ *for a request for the messages of* nid *can always be detected.*

Proof. Let us assume that an adversary returns an incomplete set of correct proofs $\mathsf{proofs} = (\mathsf{proof}_0, \mathsf{proof}_1, ..., \mathsf{proof}_n)$ for nid, and index is the index in the tree that proof_0 is located at.

If an omitted message for nid has an index lower than index, then proof_0 will contain a left sibling node with a maximum namespace maxNs where $\mathsf{maxNs} > \mathsf{nid}$, thus proving that there is an omitted message to the left of the proof set.

If an omitted message for nid has an index higher than $\mathsf{index} + n$, then proof_n will contain a right sibling node with a minimum namespace minNs where $\mathsf{nid} > \mathsf{minNs}$, thus proving that there is an omitted message to the right of the proof set.

□

5.5.4 DoS-resistance

In the design of LazyLedger, consensus nodes are not responsible for validating transactions, and thus an adversarial client may submit many invalid transactions for namespaces, forcing clients to download many invalid transactions. In a permissioned system, consensus nodes can choose which clients can submit transactions. However in a permissionless system, there ought to be a way to prioritise transactions and to make it expensive to conduct DoS attacks.

Transaction Fees

Consensus nodes can choose to prioritise transactions that include transaction fees. However, any transaction fee system should ideally not require client nodes that read messages from the application namespaces they are interested in, to also validate the application that implements the currency that transaction fees are paid in.

To achieve this, when a message is submitted to consensus nodes for inclusion in a block, the submitter of the message can also submit a ‘fee transaction’ for the currency application, and also attach to the fee transaction the hash of the ‘child’ message that the fee is paying for, such that the fee in this special fee transaction can only be collected if the message behind the specified hash is included in the same block, according to the logic of the currency application.

Client nodes of the original application whose message that the fee is paying for do not need to validate the fee transactions in the currency application; only client nodes of the currency application (*e.g.,* the consensus nodes) do. Additionally, the client nodes of the currency system application do not have to download the child message itself to verify that it has been included in the block and thus the fee has been earned, but simply verify a Merkle proof that the hash of the child message is included in the same block.

We assume that fee transactions only specify one dependency message for simplicity, but they may specify multiple dependency messages.

There does not need to be a native currency to the system, as consensus nodes can choose to accept transaction fees in any currency application that they choose to recognise.

Maximum Block Size

A maximum block size can be implemented without requiring nodes to download the entire block's data to verify that it is below a certain size. Instead, each message, and thus each leaf in the Merkle tree of messages, may have a maximum size such that if a message x is bigger than the allowed size, $\mathsf{nsHash}(x)$ would return an error, so $\mathsf{root}(M_i) = \mathsf{mRoot}_i$ and thus $\mathsf{blockValid}(h_i)$ would return false. If larger message sizes are required, a message could be chunked into multiple messages and parsed back into a single message by clients.

5.6 Implementation and Performance

We implemented a prototype of LazyLedger in 2,865 lines of Golang code. The code has been released as a free and open-source project.[1] We performed all measurements on a laptop with an Intel Core i5 2.60GHz CPU and 12GB of RAM. The prototype was not networked, as the underlying consensus protocol is arbitrary—the core contribution of LazyLedger concerns the methodology for block verification.

[1]`https://github.com/musalbas/lazyledger-prototype`

Therefore the graphs we present with regards to the amount of data downloaded are deterministic and do not contain error bars.

The prototype implements the LazyLedger block validity rules, and block data structure using namespaced Merkle trees. We also implemented (and released) several example applications using LazyLedger. Each application's state is implemented as one or more key-value stores that can be read from or modified. Applications include:

- A currency application where clients publish messages that are transactions for the transfer of funds between addresses that are elliptic curve public keys. Transactions are signed by the public keys of senders, and specify the amount of funds to send and the recipient address. In the key-value store, keys are public keys, and values are the corresponding balance of each public key, which is updated after each valid transaction.

- A name registration application where clients can: (*i*) send a balance top-up transaction to the registrar's public key using a dependency currency application, so that clients can pay for name registrations using their balance with the registrar; and (*ii*) send a registration transaction to register a specified name to their public key, which reduces the balance of the registrant, if their balance is sufficient. The registration application has one key-value store representing the in-app topped-up balance of each public key, and another key-value store where each key represents a registered name and each value represents the public key the name has been registered to.

- A dummy application for testing purposes which adds arbitrary sized specified key-value pairs to its key-value store.

We present an evaluation of LazyLedger's performance and scalability properties in relation to the goals presented in Section 6.3.3, in particular performance-related Goals 2 and 4.

Figure 5.3 compares how much data needs to be downloaded to execute the Simple Validity Rule and the Probabilistic Validity Rule to verify data availability, for varying block sizes. As expected, there is a linear relationship between block size and data downloaded for the Simple Validity Rule, as this requires downloading all of the block data to ensure that it is available. However, we can see that the relationship between block size and data downloaded for the Probabilistic Validity Rule is sub-linear and almost flat. This is because in order to execute the Probabilistic Validity Rule, nodes download a fixed number of samples and their corresponding Merkle proofs whose sizes increase logarithmically with the size of the block, as well as set of $2\sqrt{n}$ row and column Merkle roots for the block where n is the size of the block.

Recall that Goal 2 in Section 6.3.3 is application message retrieval partitioning. This means it should be possible to download all the messages for an application without needing to download the messages of other applications. Figure 5.4 compares the response size of queries for a specific namespace to a storage node ("application proofs"), for varying amounts of messages of different namespaces (measured by total bytes) that are not relevant to that query. We use currency application messages as the relevant queried namespace (although any other application could be used), fixing the number of currency messages in the block to 10, but increasing the total size of dummy application messages. We can observe that for both simple blocks and probabilistic blocks, the size of the application proofs for the relevant application only increases logarithmically, because although messages that are not in the relevant namespace do not need to be downloaded, the size of the Merkle proofs for those messages increase logarithmically as the number of total messages in the block increases. The size of the application proofs for probabilistic blocks are smaller because a two-dimensional erasure code is used, where each column and row gets its own Merkle tree, and thus the Merkle proofs are smaller because there are less items in each tree.

Recall that Goal 4 in Section 6.3.3 is application state sovereignty. This means that it should be possible for a client to determine the state of the application without requiring state from other applications that are not dependencies. Figure 5.5 follows the same setup as Figure 5.4, however instead of comparing the size of the applications proofs for the currency application, we compare the size of the state that needs to be stored by users of the relevant application (in this case, the currency application). As expected, we observe that as the size of the state of other applications increase, the size of the state that needs to be stored for the currency application remains constant.

Figure 5.6 and Figure 5.7 illustrate how the size of application proofs may vary for an application that has a dependency application. In this case we use the name registration application as an example, which requires users to follow the state of a currency application so that balance top-up transactions to the registrar can be verified. In the two graphs, we setup multiple instances of the name registration application for multiple registrars, but the user is only interested in following one of them. In Figure 5.6 we can observe that as the number of top-up transactions for the irrelevant name registration applications increase, the size of the application proofs for the relevant name registration application increases linearly, because the user must also download application proofs for the currency application, which has transactions being added to it by users of the other name registration applications. This extreme case where there are only top-up transactions defeats any scalability gains of LazyLedger, since all transactions require transactions in dependency applications that other users may follow.

However, Figure 5.7 shows the same but in the case of name registration transactions instead of balance top-up transactions. Here we see that irrelevant name registration transactions do not linearly increase the size of application proofs that need to be downloaded for other users, because only users of the relevant name registration application need to have knowledge of the registered names, and no dependency application is impacted.

5.7 Conclusion

We have presented and evaluated LazyLedger, a unique blockchain design paradigm where the base layer is only used a mechanism to guarantee the availability of on-chain messages, and transactions are interpreted and executed by end-users. We

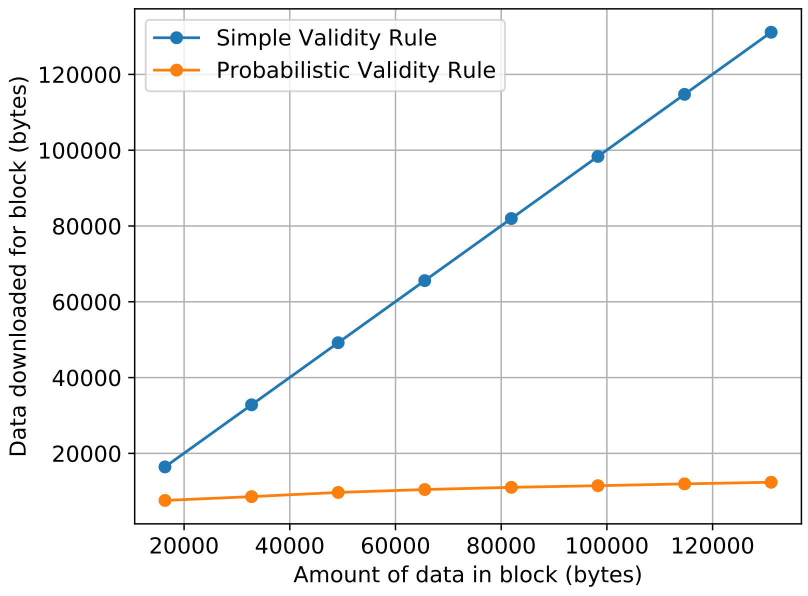


Figure 5.3: How much data needs to be downloaded to execute the block validity rule to validate data availability versus the size of the block. For the Probabilistic Validity Rule, 15 samples are used.

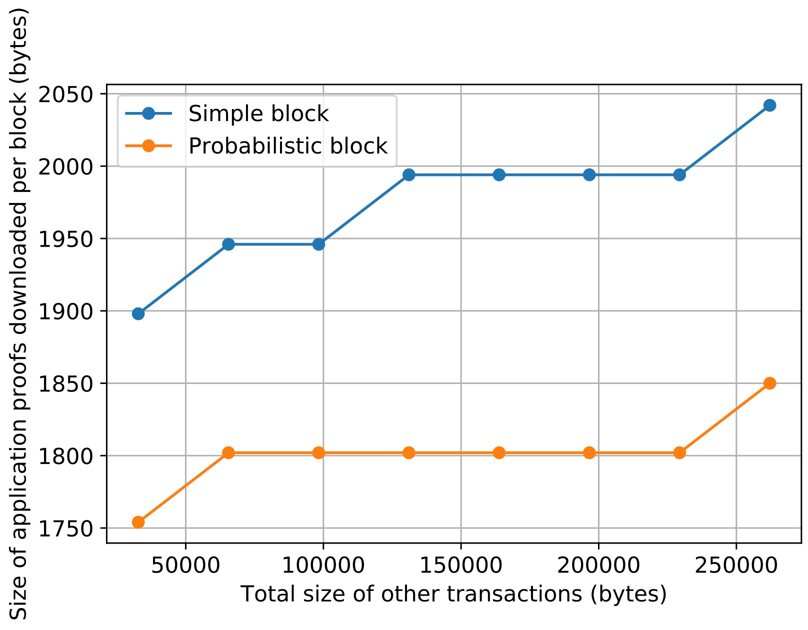


Figure 5.4: The sizes of application proofs in a block for a currency application with 10 transactions versus the total size of all of the other transactions in the block.

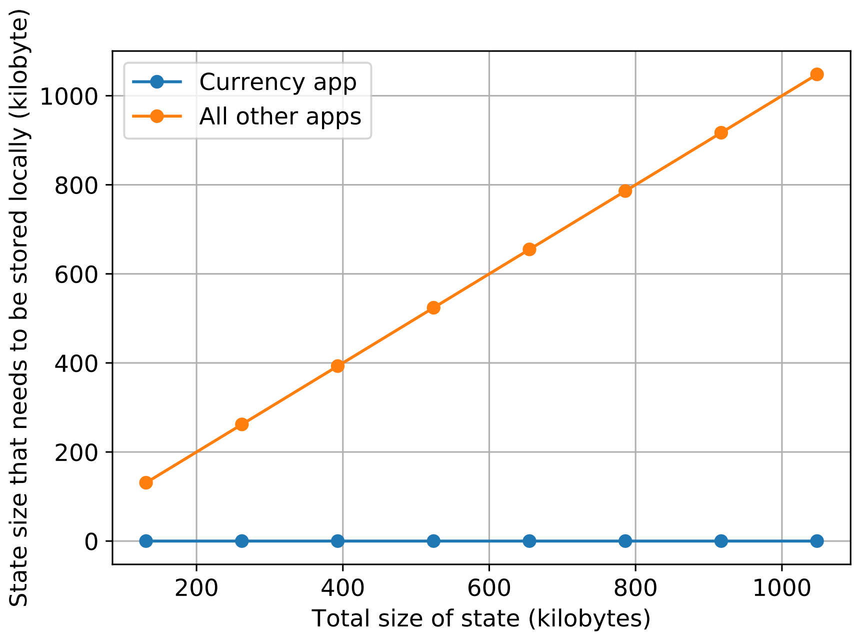


Figure 5.5: The sizes of the state that needs to be stored after a block versus the total size the state of all apps in the block, for a currency app and all other apps.

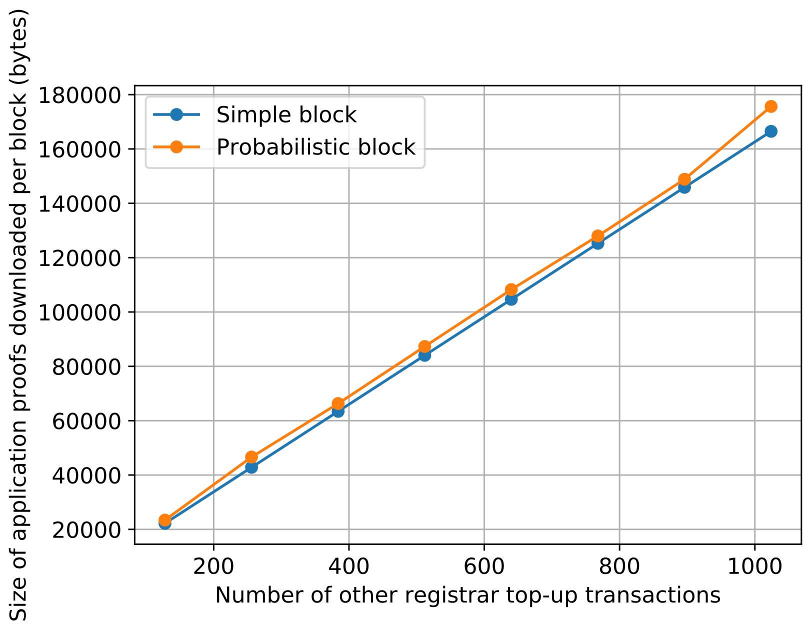


Figure 5.6: The sizes of application proofs in a block for an instance of a registrar application with 10 top-up transactions versus the number of top-up transactions for other registrar application instances in the block.

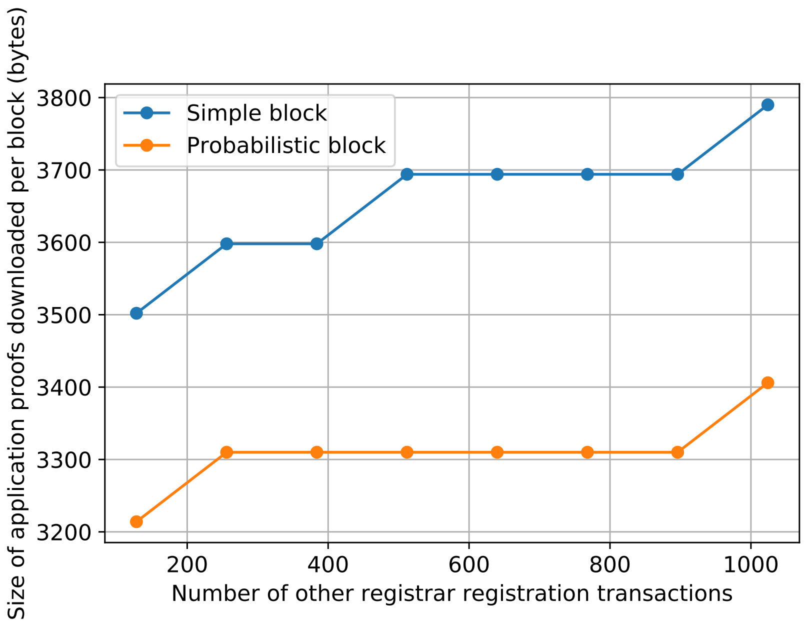


Figure 5.7: The sizes of application proofs in a block for an instance of a registrar application with 10 registration transactions versus the number of registration transactions for other registrar application instances in the block.

have shown that by reducing block verification to data availability verification, blocks can be verified in sub-linear time. Additionally, using the notion of application state sovereignty, we have shown that multiple sovereign applications can use the same chain for data availability, with only limited impact to the workload of each other's users.

Chapter 6

Contour: A Practical System for Binary Transparency

Should governments continue to encounter impediments to lawful access to information necessary to aid the protection of the citizens of our countries, we may pursue technological, enforcement, legislative or other measures to achieve lawful access solutions.

The Attorneys General and Interior Ministers of the United States, the United Kingdom, Canada, Australia and New Zealand (Five Eyes)

6.1 Introduction and Motivation

Blockchain-based systems such as Bitcoin make the history of their ledgers transparent, in order to allow all participants to confirm for themselves whether or not transactions included in blocks are complying with the protocol rules, thus removing the need for a centralised or single trusted third party. However, the idea of transparency is also useful for holding centralised third parties accountable for their actions, by allowing users to audit their actions in a blockchain-based ledger that acts as an audit log, even if that log can only be written to by a centralised party. By utilising the append-only property of a blockchain log (see Section 2.2), the centralised party should not be able to hide or erase evidence of misbehaviour.

One of the technical settings in which the idea of transparency has been most thoroughly deployed is the issuance of X.509 certificates; this is likely due partially to the nature of these certificates (which are themselves intended to be globally visible), and partially to the many publicised failures of major certificate authorities (CAs) [71, 96]. A long line of recent research [94, 22, 84, 129, 105] has provided and analysed solutions that bring transparency to the issuance of both TLS certificates ("certificate transparency") and to the assignment of public keys to end users ("key transparency").

Despite their differences, many of these systems share a fundamentally similar architecture [44]: after being signed by CAs, certificates are stored by *log servers* in a globally visible append-only log. Clients do not accept certificates unless they have been included in such a log, and are thus publicly auditable. To determine this they rely on *auditors*, who are responsible for checking inclusion of the specific certificates seen by clients. Because auditors are often implemented as software running on the client (*e.g.,* a browser extension), they must be able to operate efficiently. Finally, in order to expose misbehavior, *monitors* (inefficiently) audit the certificates stored in a given log to see if they satisfy the rules of the system.

To achieve the append-only log property where the log server is centralised (see Section 2.3.2 for background on centralised consensus), an additional line of communication is needed between the auditor and monitor in the form of a *gossip* protocol [113, 46]. In such a protocol, the auditor and monitor exchange information on their current view of the log (*e.g.,* the latest block headers), which allows them to detect whether or not their views are *consistent*, and thus whether or not the log server is misbehaving by presenting "split" views of the log. If such attacks are possible, then the accountability of the system is destroyed, as a log server can present one log containing all certificates to auditors (thus convincing it that its certificates are in the log), and one log containing only "good" certificates to monitors (thus convincing them that all participants in the system are obeying the rules). This is identical to an adversary forking a blockchain as described in Section 2.2.2, and showing different users different sides of the fork.

While gossiping can detect this misbehaviour, it is ultimately a retroactive mechanism — i.e., it detects rather than prevents this misbehavior — and is thus most effective in settings where no man-in-the-middle (MitM) attack can occur, so the line of communication between an auditor and monitors remains open. Various systems have been proposed recently that do not rely on retrospective gossiping to deal with split view attacks, such as Collective Signing [143], but perhaps the most prominent example of such a system is Bitcoin and other decentralised blockchain-based systems. The decentralised consensus in Bitcoin and other decentralised blockchains makes forking the chain and conducting split view attacks economically expensive even in the presence of man-in-the-middle (MitM) attacks, or 'eclipse' attacks where the attacker fully controls a client's connections with other nodes. We analyse the economic costs in Section 6.5.1.

Because of the effectiveness of these approaches, there has been interest in repurposing them to provide not only transparency for certificates or monetary transfers, but for more general classes of objects ("general transparency") [59]. One specific area that thus far has been relatively unexplored is the setting of software binary distribution ("binary transparency"). Bringing transparency to this setting is increasingly important, as there are an increasing number of cases in which actors target devices with malicious software signed by the authoritative keys of update servers. For example, the Flame malware, discovered in 2012, was signed by a rogue Microsoft certificate [71] and masqueraded as a routine Microsoft software update [111]. In 2016, a US court compelled Apple to produce and sign custom firmware in order to disable security measures on a phone that the FBI wanted to unlock [64].

Aside from its growing relevance, binary transparency is particularly in need of exploration because the techniques described above for both certificate transparency and Bitcoin cannot be directly translated to this setting. Whereas certificates and Bitcoin transactions are small (on the order of kilobytes), software binaries can be arbitrarily large (often on the order of gigabytes), so cannot be easily stored and replicated in a log or ledger. Most importantly, by their very nature software packages have the ability to execute code on a system, so malicious software packages can easily disable gossiping mechanisms or launch wider (and long-lasting)

MitM attacks. This makes retroactive methods for detecting misbehaviour almost uniquely poorly suited to this setting, in which clients need to know that a software package is auditable by independent parties *before* installing it, not after.

In this chapter, we make the following contributions:

- We present the design of Contour, a solution for binary transparency that utilizes the Bitcoin blockchain in an efficient manner to proactively prevent clients from installing malicious software, even in the face of long-term MitM attacks and malicious binaries.

- We analyse both the security and efficiency of Contour. Given the volume of related research on certificate transparency, we also present some comparisons here, and argue that ours is the first efficient solution to provide these security guarantees without requiring any coordination cost, in the form of selecting a central entity to perform authorization, or otherwise forming a Sybil-free set of log servers. This ensures that Contour could be easily and securely deployed today.

- To validate our efficiency claims, we describe an implementation of Contour and benchmark its performance, finding that almost all operations can be performed very quickly (on the order of microseconds), that auditors can store minimal information (on the order of kilobytes), and that arbitrary numbers of binaries can be represented by a single small (235-byte) Bitcoin transaction.

- We also validate our claims of real-world relevance by presenting the application of Contour to the current package repository for the Debian operating system. We find that it would require minimal overhead for the existing actors within this system, and cost under 2 USD per day.

6.2 Related Work

As mentioned in the introduction, there is a significant volume of related work on the idea of transparency, particularly in the settings of certificates and keys. We describe the most relevant work here.

6.2.1 Certificate Transparency

RFC 6962

RFC 6962 [94] is the protocol standard for TLS certificate transparency adopted by web browsers including Google Chrome and Mozilla Firefox. Clients such as web browsers use a built-in of log servers, which they require valid certificates to be included in. Auditors verify Merkle proofs that certificates are included in logs before clients accept them. Monitors—which may be website operators that want to protect their users from rogue certificates—read the log servers to look for misbehaviour.

To allow servers to deploy TLS certificates before they are included in a log, log servers produce a *Signed Certificate Timestamp* when a certificate is submitted to a log. This is a signed promise by a log to include a certificate within a certain amount of time. This allows clients to accept certificates before they are included in the log. If a log server fails to honour its promise then the signature can be used as cryptographic evidence of this.

Chase and Meiklejohn abstract certificate transparency into the general idea of a "transparency overlay" [44] and prove its security.

Accountable Key Infrastructure (AKI) and Attack Resilient Public Key Infrastructure (ARPKI)

AKI [84] adopts the log server concept, but addresses shortcomings in RFC 6962. As RFC 6962 does not support certificate revocation, Merkle audit proofs for certificates in the log are valid indefinitely. To address this, AKI's "integrity tree" commits to a list of all of the certificates in the system in each log update, ordered by domain name. If a certificate is revoked, then the next log update will remove the certificate from the list. This allows for auditors to verify Merkle non-inclusion proofs to verify that a certificate is not in the latest version in the log. Similar approaches for handling revocation transparency have also been proposed for RFC 6962, using a sparse Merkle tree to commit to the set of certificates in the log [129, 93].

AKI also allows domain owners to define a specific set of certificate authorities that are authorised to sign certificates on for the domain, a minimum defined threshold of which is required to sign a certificate for it to be valid. This is in contrast to the current Public Key Infrastructure for TLS, where any one certificate authority in the client's root of trust is authorised to sign certificates for any domain. In AKI, validators (which are akin to monitors) check the logs for misbehaviour.

ARPKI [22] builds on AKI, augmenting it so that the domain owner selects two certificate authorities and one log server that validate each other's actions, so that a rogue certificate can only be created if all three actors have been compromised.

6.2.2 Key Transparency

CONIKS

CONIKS [105] is a key transparency protocol aimed for end-to-end encrypted communication software. It relies on identity providers with their own namespaces, which provide key directories with name-to-key bindings. Users can query identity providers to get the key of a user. Each identity provider also operates its own log of keys, similar to certificate transparency log servers, so that malicious name-to-keys bindings can be detected by users. It similarly relies on gossiping to detect log equivocation by log servers.

EthIKS [32] proposes an implementation of a CONIKS auditor as an Ethereum smart contract, so that users can trust the Ethereum network to audit CONIKS logs for consistency and non-equivocation.

ClaimChain

ClaimChain [90] is a key transparency protocol developed in the context of end-to-end encrypted email. Like CONIKS and certificate transparency, it also makes use of append-only logs. However rather than having a set of centralised log servers, each ClaimChain user operates their own log known as a ClaimChain. Each user's log records claims about their own keys, as well beliefs about other the keys of other users, which are recorded as cross-references to other chains. It relies on gossiping to detect log equivocation.

Certcoin

Certcoin [67] propose a blockchain-based decentralised public key infrastructure for domain names that eliminates the need for certificate authorities. It works by allowing users to register domains directly on the blockchain, and specify the public key associated with the domain at the point of registration. This creates a domain-to-key mapping that does not require certificate authorities.

Smart Contract Public Key Infrastructure (SCPKI)

SCPKI [2] is a public key infrastructure for user identities designed as a smart contract on the Ethereum blockchain. It uses a PGP-style web-of-trust model [69] where users can vouch for attributes associated with the keys of other users, such as name.

6.2.3 Binary Transparency

Mozilla

The Mozilla community has proposed an approach to binary transparency [21] that is layered on top of the RFC 6962 certificate transparency protocol, where the hashes of binaries are included in logged certificates. Auditors can then verify inclusion of binaries in a log by checking that certificates that include the hashes of the binaries are in the log.

As mentioned in Section 6.1 and Section 6.2.1, RFC 6962 certificate transparency is a retroactive transparency mechanism that is reliant on client-side gossiping to detect log equivocation. This approach is therefore not secure in the presence of malicious binaries that can disable gossiping in the client's auditor software, and long term MiTM attacks.

Golang

Golang implements a binary transparency protocol for Go packages [48]. It is based on Trillian [59], a general-purpose transparent log implementation. Like the Mozilla community's binary transparency proposal, it also relies on gossiping to detect log equivocation and is therefore also not secure in the presence of malicious binaries that can disable gossiping, and long term MiTM attacks.

Collective Signing (CoSi)

CoSi [143] is a protocol that aims to keep authorities honest, by using a decentralised network of witnesses that collectively sign messages (such as certificates) produced by authorities, to confirm that they are validated and/or logged. It uses a tree-based protocol to allow thousands of witnesses to collectively generate a Schnorr multisignature [114] that can be efficiently verified.

This is a proactive transparency mechanism, and is thus suitable for a binary transparency use case. However CoSi does not address the problem of how the set of witnesses, which must be Sybil-free, should be selected and maintained. This is left as an open problem. This creates a deployment challenge that we wish to avoid in Contour, by using an existing consensus network—Bitcoin.

Chainiac

Chainaic [112] is a binary transparency protocol that is based on CoSi, developed concurrently with Contour. It uses a set of CoSi witnesses who verify that software updates conform to release policies. Additionally, it proposes an append-only data structure called a skip-chain, which allows clients efficiently navigate update timelines, both forwards (for upgrades) and backwards (for downgrades).

6.2.4 General Transparency

Catena

Catena [146] is a Bitcoin-based general transparency protocol developed concurrently with Contour. While both Catena and Contour utilise similar features of Bitcoin to achieve efficiency, they differ in their focus (general vs. binary transparency), and thus in the proposed threat model; *e.g.,* they dismiss eclipse attacks [136] on the Bitcoin network, whereas we consider them well within the scope of a MitM attacker.

Trillian

Trillian [59] is a general-purpose implementation of a Merkle tree-based log, that can be used to implement log servers that stores auditable data. The log servers are centralised entities that produce signed tree heads, and thus gossiping is necessary to detect equivocation.

6.2.5 Reproducible Builds

Reproducible builds [112, 30] is a technology that allows an interested user to verify that a certain software binary was built from a given source code. This is achieved by providing a deterministic script for building the software, so that it always returns the same binary. While reproducible builds allow a user to verify that a binary matches a certain source code, it does not allow users to verify that the binary–or source code–is the same that was given to all other users, or is publicly auditable. Therefore, binary transparency and reproducible builds can be used to complement each other; reproducible builds can be used to show that a source code matches a binary, and binary transparency can be used to ensure that a record of the source code and binary is publicly auditable (*i.e.,* the source code could be packaged with the binary).

6.3 Threat Model and Setting

In this section, we describe the actors in our software binary transparency system (Section 6.3.1), along with the interactions between these actors (Section 6.3.2), and the goals we aim to achieve (Section 6.3.3).

6.3.1 Participants

We consider a system with five types of actors: services, authorities, monitors, auditors, and clients. We describe each of these types below in the singular, but for the correct and secure functioning of a transparency overlay we require a distributed set of auditors and monitors, each acting independently.

Service: The service is responsible for producing actions, such as the issuance of a software update. In order to have these actions authorized, they must be sent to the *authority*.

Authority: The authority is responsible for publishing *statements* that declare it has seen a given action taken by a service; e.g., that it has been sent a given software binary. These statements furthermore claim that the authority has—

in some form — published these actions in a way that allows them to be inspected by the *monitor*. As this inspection is typically inefficient, the authority is also responsible for placing its statements into a public *audit log*, where they can be efficiently verified by the *auditor*.

Monitor: The monitor is responsible for inspecting the actions published by the authority and performing out-of-band tests to determine their validity (e.g., to ensure that software updates do not contain malware).

Auditor: The auditor is responsible for checking specific actions against the statements made by the authority that claim they are published.

Client: The client receives software updates from either the authority or the service, along with a statement that claims the update has been published for inspection. It outsources all responsibility to the auditor, so in practice the auditor can be thought of as software that sits on the client (thus making the client and auditor the same actor).

6.3.2 Interactions

In terms of the interactions between these entities, one of the main benefits of Contour — as discussed in the introduction — is that entities do not need to engage in prolonged multi-round interactions like gossiping, but rather pass messages atomically to one another. As we see in Section 6.5.1, this minimizes the advantage that an adversary could gain by launching man-in-the-middle attacks. We therefore outline only non-interactive algorithms needed to generate messages, rather than interactive protocols, and wait to specify the exact inputs and outputs until we present our construction in Section 6.4.

Authority.`commit`**:** The authority runs this algorithm to commit statements to the audit log.

Authority.`prove_incl`**:** The authority runs this algorithm to provide a proof that a specific statement is in the audit log.

Auditor.`check_incl`: The auditor runs this algorithm to check the proof of inclusion for a specific statement.

Monitor.`get_commits`: The monitor runs this algorithm to retrieve relevant commitments from the audit log.

6.3.3 Goals

We break the goals of the system down into security goals (denoted with an S) and deployability goals (denoted with a D).

In all our security goals, we aim to defend against the specified attacks in the face of malicious authorities that, in addition to performing all the usual actions of the authority, can also perform man-in-the-middle attacks on the auditor's network communications, and can compromise the client's machine with malicious software updates. If additional adversaries are considered we state them explicitly.

S1: No split views. We would like to prevent split-view attacks, in which the information contained in the audit log convinces the auditor that the authority published an action taken in the system, and thus it is able to be inspected by monitors, whereas in fact it is not and only appears that way in the auditor's "split" view of the log.

S2: Auditor privacy. We would like to ensure that the specific binaries in which the auditor is interested are not revealed to any other parties apart from the authority (where software binaries may be downloaded from), as this might reveal, for example, that a client has a software version that is susceptible to malware. We thus consider how to achieve this not only in the face of malicious authorities, but in the case in which all parties aside from the auditor are malicious.

D1: Efficiency. We would like Contour to operate as efficiently as possible, in terms of computational, storage, and communication costs. In particularly, we would like the overhead beyond the existing requirements for a software distribution system to be minimal.

D2: Minimal setup. In addition to the computational overheads, we would like as little effort — in terms of, e.g., coordination — to be done as possible in order to deploy Contour. Specifically, we would like to avoid the need to bootstrap and govern a new set of Sybil-free nodes or witnesses (such as CoSi [143]).

6.4 Design of Contour

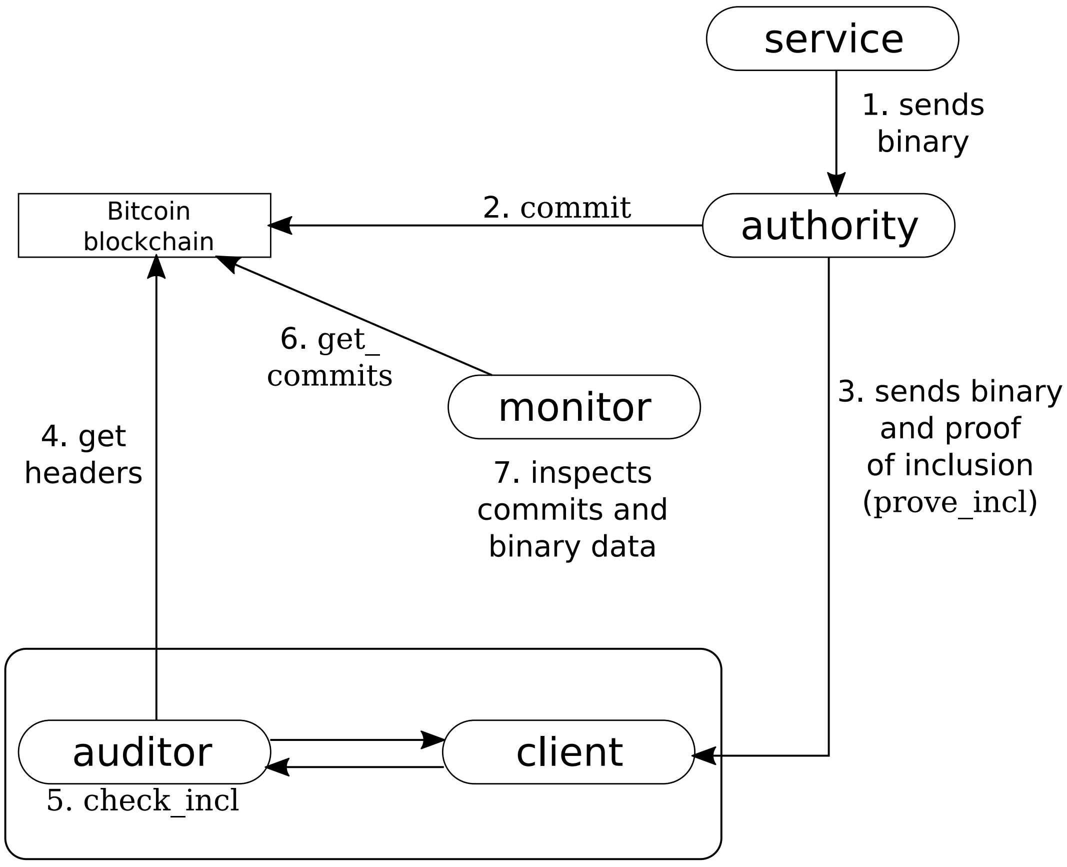


Figure 6.1: The overall structure of Contour.

In this section we describe the overall design of Contour. An overview of the interactions between all the various actors can be seen in Figure 6.1.

6.4.1 Setup and Instantiation

Contour and its security properties make use of a blockchain, whose primary purpose — as we see in Section 6.5.1 — is to provide an immutable ledger that prevents split-view attacks. Because the Bitcoin blockchain is currently the most expensive to attack, we use it here and in our security analysis in Section 6.5.1, but observe that any blockchain could be used in its place. An authority must initially establish a known Bitcoin address that Contour commitments are published with. As knowledge of the private key associated with the Bitcoin address is required to sign transactions to spend transaction outputs sent to the address, this acts as the root-of-trust for the authority. This address can be an embedded value in the auditor software. An initial amount of coins must be sent to the Bitcoin address to enable it to start making transactions from the address.

6.4.2 Logging and Publishing Statements

To start, the authority receives actions from services; i.e., software binaries from the developers of the relevant packages (Step 1 of Figure 6.1). As it receives such a binary, it incorporates its hash as a leaf in a Merkle tree with root h_T. The root, coupled with the path down to the leaf representing the binary, thus proves that the authority has seen the binary, so we view the root as a batched statement attesting to the fact that the authority has seen all the binaries represented in the tree.

`commit`(h_T)**:** Form a Bitcoin transaction in which one of the outputs embeds h_T by using `OP_RETURN`. One of the inputs must be a previous transaction output that can only be spent by the authority's Bitcoin address (i.e. a standard Bitcoin transaction to the authority's address). The other outputs are optional and may simply send the coins back to the authority's address, according to the miner's fees it wants to pay. (See Section 6.6.2 for some concrete choices.) Sign the transaction with the address's private key and publish to the Bitcoin blockchain and return the raw transaction data, denoted tx.

Crucially, the `commit` algorithm stores only the root hash in the transaction, meaning its size is independent of the number of statements it represents. Furthermore, if the blockchain is append-only — i.e., if double spending is prevented — then the log represented by the commitments in the blockchain is append-only as well.

6.4.3 Proving Inclusion

After committing a batch of binaries to the blockchain, the authority can now make these binaries accessible to clients. When a client requests a software update, the authority sends not only the relevant binary, but also an accompanying proof of inclusion, which asserts that the binary has been placed in the log and is thus accessible to monitors (Step 3 of Figure 6.1).

To generate this proof, the authority must first wait for its transaction to be included in the blockchain (or, for improved security, for it to be embedded k blocks into the chain). We denote the header of the block in which it was included as head_B. The proof then needs to convince anyone checking it of two things: (1) that the relevant binary is included in a Merkle tree produced by the authority and (2) that the transaction representing this Merkle tree is in the blockchain. Thus, as illustrated in Figure 6.2, this means providing a path of hashes leading from the values retrieved from the blockchain to a hash of the statement itself.

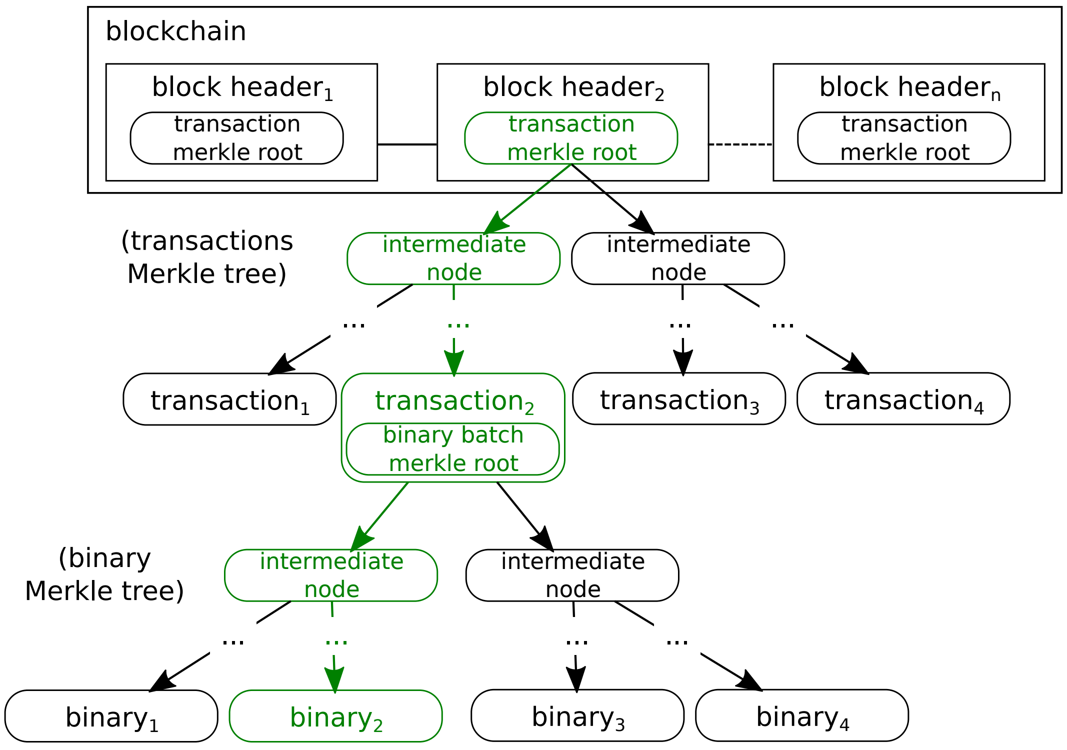


Figure 6.2: An example of a path of hashes leading from the block's transactions merkle root to the hash of the statement$_2$.

For a given binary binary, the algorithm `prove_incl` thus runs as follows:

`prove_incl`$(\mathsf{tx}, \mathsf{head}_B, \mathsf{binary})$**:** First, form a Merkle proof for the inclusion of tx in the block represented by head_B. This means forming a path from the root hash stored in head_B to the leaf representing tx; denote these intermediate hashes by π_{tx}. Second, form a Merkle proof for the inclusion of binary in the Merkle tree represented by tx (using the hash h_T stored in the `OP_RETURN` output) by forming a path from h_T to the leaf representing binary; denote these intermediate hashes by π_{binary}. Return $(\mathsf{head}_B, \mathsf{tx}, \pi_{\mathsf{tx}}, \pi_{\mathsf{binary}})$.

6.4.4 Verifying Inclusion

To verify this proof, the auditor must check the Merkle proofs, and must also check the authority's version of the block header against its own knowledge of the Bitcoin blockchain. This means that the auditor must first keep up-to-date on the headers in the blockchain, which it obtains by running an SPV client (Step 4 in Figure 6.1). By running this client, the auditor builds up a set $S = \{\mathsf{head}_{B_i}\}_i$ of block headers, which it can check against the values in the proof of inclusion. This means that, for a binary binary, `check_incl` (Step 5 in Figure 6.1) runs as follows:

`check_incl`$(S, \mathsf{binary}, (\mathsf{head}_B, \mathsf{tx}, \pi_{\mathsf{tx}}, \pi_{\mathsf{binary}}))$**:** First, check that $\mathsf{head}_B \in S$; output 0 if not. Next, extract h_T from tx (using the hash stored in the `OP_RETURN` output), form $h_{\mathsf{binary}} \leftarrow H(\mathsf{binary})$, and check that π_{binary} forms a path from the leaf h_{binary} to the root h_T. Finally, form $h_{\mathsf{tx}} \leftarrow H(\mathsf{tx})$, and check that π_{tx} forms a path from the leaf h_{tx} to the root hash in head_B. If both these checks pass then output 1; otherwise output 0.

6.4.5 Discussion of Data Availability

A malicious authority may publish Merkle roots of binaries to the Bitcoin blockchain, but not publish the data of the entire tree or the binaries. We do not propose or enforce an in-protocol mechanism to prevent this such as data availability proofs (Section 4.5). This is because the role of monitors is social, as when misbehaviour is detected, any reaction to the misbehaviour occurs outside the protocol, within the relevant community. For example, there may not be an objective mechanism to determine that a binary contains malicious code (is it malicious code or is it a feature?). Therefore as mentioned in Section 6.3, the actions of monitors are out-of-scope to this chapter.

While data unavailability would prevent monitors from being able to inspect specific binaries, they would still be aware that they have been unable to download the binaries, which is an indication of misbehaviour that can raised to the community. If no monitor is able unable to download the data, they may conclude that the authority has not published the data.

In order to give auditors data availability guarantees, authorities may publish binaries to a data availability layer such as LazyLedger (Chapter 5), however we consider this to be out-of-scope as do not enforce an in-protocol data availability scheme for the reasons above.

6.5 Evaluation

In this section, we evaluate Contour in terms of how well it meets the security goals (Section 6.5.1) and deployability goals (Section 6.5.2) specified in our threat model in Section 6.3.3.

6.5.1 Security goals

No split views (S1)

In order to prevent split views, we rely on the security of the Bitcoin blockchain and its associated proof-of-work-based consensus mechanism. If every party has the same view of the blockchain, then split views of the log are impossible, as there is a unique commitment to the state of the log at any given point in time. The ability to prevent split views therefore reduces to the ability to carry out attacks on the Bitcoin blockchain. We break down the costs of such attacks below: in particular, we first consider the cost of mining a single block, and then separately examine the case when the adversary can carry out an eclipse attack — in which, recall from Section 2.3.3, it can control the auditor's view of the blockchain — and the case when it cannot.

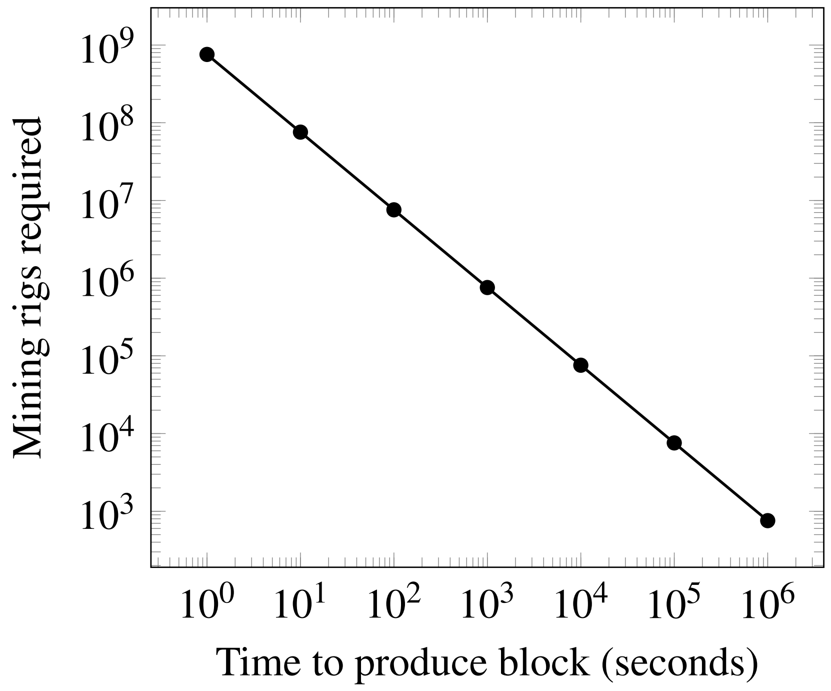


Figure 6.3: The number of Antminer S17+ rigs required to produce blocks under a certain time limit.

Cost to mine a single block. The probability of a miner finding a valid block after each hashing attempt is $\frac{2^{16}-1}{2^{48}D}$, where D is the periodically adjusted difficulty of the network. For a miner to mine a block then, they must make on average $\frac{2^{48}D}{2^{16}-1}$ hashing attempts. The total electricity cost (C) of mining a block is thus

$$C = \frac{2^{48}D}{2^{16}-1} \cdot J \cdot E \qquad (6.1)$$

where J is the number of joules required per hashing attempt, and E is the electricity cost of one joule. As of December 2019, the most energy-efficient Bitcoin mining hardware is the Antminer S17+, which has an energy cost of $4 \cdot 10^{-11}$ joules per hash, and the average retail price of one kilowatt hour in the US is 0.10 USD [149]. The cost per joule, E, is therefore $\frac{0.10}{1000 \cdot 60 \cdot 60} = 2.8 \cdot 10^{-8}$ USD. As of December 2019, the Bitcoin mining difficulty (D) is 12,876,842,089,682. Plugging these numbers into Equation 6.1, the total electricity cost to mine a block under these assumptions, using the most efficient hardware and assuming standard electricity costs, is thus 61,450 USD.

To also take account of hardware costs, we observe that the number of mining rigs N needed to mine a block in S seconds is

$$N = \frac{(\frac{2^{48}D}{2^{16}-1})}{H \cdot S} \tag{6.2}$$

where H is the number of hashes that the mining rig is capable of calculating per second. This formula is graphed in Figure 6.3 for the Antminer S17+ rig, which is capable of calculating 73 terahashes per second and has a retail cost of 1,930 USD [82]. We use these formulas to estimate the cost of split-view attacks in the following analysis.

Using eclipse attacks. If an eclipse attack is possible, the adversary can “pause” the auditor at a block height representing some previous state of the log, and can prevent the auditor from hearing about new blocks past this height. It is then free to mine blocks at its own pace, and can thus launch a successful split-view attack solely by mining k blocks, where k is the number of blocks the auditor requires to be mined after a block containing a given commitment in order to consider that commitment as valid. (It is standard in most Bitcoin wallets to use $k = 6$.)

Using our rough estimates above, it would cost the adversary 61,450 USD in electricity costs to mine a block, or 368,700 USD for $k = 6$. The hardware costs depends on how much time the adversary needs to conduct the attack, or how long they are able to continue their man-in-the-middle attack on the auditor. If — as a conservative number — the adversary wants to conduct the attack within a week, it must mine a block every 1.4 days to produce 6 blocks, which requires 6,263 mining rigs at a hardware cost of 12,087,590 USD. This brings the total cost of the attack to 12.5M USD, which is likely to deter at least a large fraction of potential adversaries. It may be affordable for a powerful adversary, however, but the attack is also fundamentally targeted: if the adversary wants to later compromise previously non-eclipsed auditors, it must mine a new set of blocks (assuming these auditors have more up-to-date blocks) and pay the electricity costs again. Even for an adversary with few financial constraints, this makes it significantly more difficult to conduct such an attack on a wide scale.

Furthermore, if the adversary takes 1.4 days to mine a block, or in general the auditor sees no new blocks until long after the expected 10-minute interval, it may assume that an eclipse attack is being performed. We can thus greatly increase the cost of the attack by adding simple checks to the auditor to ensure that there is a maximum interval between blocks. If we generously set such a check to require a maximum of 6 hours between blocks, then a total of 35,074 mining rigs are required at a cost of 67.7M USD.

In addition, the blocks must still follow the same difficulty level as honest blocks, so by mining these only in the eclipsed view of the network the adversary is not only expending the energy needed to do so but is also forfeiting the mining reward associated with them. As of December 2019, the Bitcoin mining reward is 12.5 bitcoins, or roughly 87,500 USD at 7000 USD per coin [70], so for $k = 6$ the adversary must additionally forfeit 525,000 USD.

Ignoring eclipse attacks. If, for whatever reason, an eclipse attack is not possible, then an adversary can perform a split-view attack only if it can fork the Bitcoin blockchain. This naïvely requires it to control 51% of the network's mining power.

As of December 2019, the total hashing power of the Bitcoin network was 100,497,473 terahashes per second.[1] Conducting a 51% attack would therefore require the adversary to be able to compute more than 100,497,473 terahashes per second. Per hour, the total electricity cost would be $100497473 \cdot 10^{12} \cdot 3600 \cdot J \cdot E$, or — using our earlier estimates for J and E — 401,990 USD per hour. In terms of hardware costs, if we use the figures for the Antminer S17+ from before, the total number of mining rigs required would be greater than $\frac{100497473 \cdot 10^{12}}{73 \cdot 10^{12}} = 1376677$, at a total cost of at least 2657M USD.

While more sophisticated attacks, such as selfish mining [63], have proposed strategies that fork the blockchain (also known as a "double spend" attack) using only 25% of the mining power, this would still require an investment of hundreds of millions of dollars. Such an attack would furthermore be highly visible, as the blockchain is regularly monitored for forks.

[1]`https://www.blockchain.com/en/charts/hash-rate`

Operation	Time complexity
`commit`	$O(n_S)$
`prove_incl` (one-time)	$O(\log(n_T))$
`prove_incl` (per statement)	$O(\log(n_S))$
`check_incl`	$O(\log(n_S) + \log(n_T))$

Table 6.1: Asymptotic computational costs for the operations of Contour, where n_S is the number of statements in a batch and n_T is the number of transactions in a block.

Auditor privacy (S3)

Recall from Section 6.3.2 that one of the goals of Contour was to avoid prolonged interactions and engage only in the atomic exchange of messages. In particular, the auditor receives pre-formed proofs of inclusion from the authority (as opposed to requesting them for specific statements), retrieves commitments directly from the blockchain and does not engage in any form of gossip with monitors. We thus achieve privacy by design, as at no point in the process does the auditor reveal the statements in which it is interested to any other party.

One particular point to highlight is that Contour achieves auditor privacy despite the fact that auditors run SPV clients, which are known to potentially introduce privacy issues due to the use of Bloom filtering and the reliance on full nodes. This is because the proofs of inclusion contain both the raw transaction data and the block header, so the auditor does not need to query a full node for the inclusion of the transaction and can instead verify it itself (and, as a bonus, saves the bandwidth costs of doing so).

6.5.2 Deployability goals

Efficiency (D1)

Table 6.1 summarizes the computational complexity of each of the operations required to run Contour, and Table 6.2 summarizes the size complexity.

Object	Size Complexity
Inclusion proof	$O(\log(n_S)+\log(n_T))$
Log commitment (tx)	$O(1)$

Table 6.2: Asymptotic storage costs for the objects in Contour, where n_S is the number of statements in a batch and n_T is the number of transactions in a block.

We observe that the end-user devices on which the auditor is run impose a relatively minimal performance overhead (with everything logarithmic in n_S and/or n_T), as this simply involves verifying Merkle proofs of transactions or statements. Authorities have linear performance overhead with respect to the number of updates in each batch, as this involves computing the root of a Merkle tree. We confirm this in our implementation in Section 6.6.2.

Minimal setup (D2)

In terms of coordination, the only setup requirement in Contour is the authority generating a key to be used as the root-of-trust in auditor software, as the rest is a matter of adding software to existing actors in software distribution ecosystems. We do not require the authority to bootstrap and maintain a Sybil-free set of nodes or witnesses, and we instead rely on the existing Bitcoin blockchain for non-equivocation.

6.6 Implementation and Performance

To test Contour and analyze its performance, we have implemented and provided benchmarks for a prototype Python module and toolset that developers can use. We have released the implementation as an open-source project.[2]

6.6.1 Implementation details

The implementation consists of roughly 1000 lines of Python code, and provides a set of developer APIs and corresponding command-line tools. We used SHA-256 as the hashing algorithm to build Merkle trees, and modified versions of an existing

[2] `https://github.com/musalbas/contour`

Merkle tree implementation[3] and a Python-based Bitcoin library `pycoinnet`[4] in order to develop our Merkle tree and SPV client, respectively.[5]

Authority: We provide API calls for Authority.`commit`, which commits batches of statements to the Bitcoin blockchain, and Authority.`prove_incl`, which allows it to generate inclusion proofs for individual statements.

Auditor: We provide an API call for Auditor.`check_incl`, which allows end-user software to verify proofs of inclusion. We also provide an Auditor.`sync` call that uses the Bitcoin SPV protocol to download and verify all the block hashes in the Bitcoin blockchain, so that inclusion proofs can be efficiently verified independently of third parties. (This call needs to be run only once.)

Monitor: We provide an API call for Monitor.`get_commits`, which gets all the statement batches associated with a specific authority. Monitors can then use these commitments to check the validity of the statement data (which they can retrieve from the authority or an archival node using a web server), and do whatever manual inspection is necessary; we consider this functionality outside of the scope of this paper.

6.6.2 Performance

To evaluate the performance of our implementation, we tested all the operations listed above on a laptop with an Intel Core i5 2.60GHz CPU and 12GB of RAM, that was connected to a WiFi network with an Internet connection of 5Mbit/s. We also assume that a batch to be committed contains 1 million statements, although as was seen in Table 6.1 — and confirmed later on in Figure 6.4 — these numbers scale as expected (either logarithmically or linearly), so it is easy to extrapolate the results for other batch sizes given the ones we present here.

[3]`https://github.com/jvsteiner/merkletree`

[4]`https://github.com/richardkiss/pycoinnet`

[5]The Merkle tree modifications were necessary because the Merkle tree implementation in Bitcoin has a documented bug that must, for consensus reasons, be replicated in software using the Bitcoin protocol. The `pycoinnet` modifications were necessary as this library provides only the code to communicate using the Bitcoin protocol, rather than connecting to Bitcoin itself.

Operation	Time (µs)	σ (µs)
`commit`	**5.93 (s)**	**0.297 (s)**
`prove_incl` (one-time)	8.5	5.4
`prove_incl` (per statement)	12	6.4
`check_incl`	224	62.14

Table 6.3: Average time of individual operations, and standard deviation σ, where n is the number of statements in a batch. The timings for `commit` were averaged over 20 runs, and for `prove_incl` and `check_incl` over 1M runs. The timings for `commit` are in bold to emphasize that they are in seconds, not microseconds.

We consider the complexity of these operations in terms of their computational, storage, and bandwidth requirements. A summary of our timing benchmarks can be found in Table 6.3.

Number of transactions per block

The overhead of both generating and verifying a proof of inclusion is dependent on the number of transactions in a Bitcoin block. To capture the worst-case scenario, we consider the maximum number of transactions that can fit into a block. The Bitcoin block size limit is 1 MB (excluding SegWit), up to 97 bytes of which is non-transaction data. The minimum transaction size is 166 bytes, so the upper bound on the number of transactions in a given block is 6,023. While this is far higher than the number of transactions that Bitcoin blocks usually contain, we nevertheless use it as a worst-case cost and an acknowledgment that Bitcoin is evolving and blocks may grow in the future.

Authority overheads

To run `commit` and `prove_incl`, an authority must have access to the full blocks in the Bitcoin blockchain, as well as the ability to broadcast transactions to the network. Rather than achieve these by running the authority as a full node, our implementation uses external blockchain APIs supplied by `blockchain.info` and `blockcypher.com`. This decision was based on the improved efficiency and ease of development for prototyping, but it does not affect the security of the system: authorities do not need to validate the blockchain, as invalid blocks from a dishonest external API simply result in invalid inclusion proofs that are rejected by the auditor.

To run `commit`, an authority must first build the Merkle tree containing its statements. Sampled over 20 runs, the average time to build a Merkle tree for 1M statements was 5.9 s ($\sigma = 0.29$ s). After building the tree, an authority next embeds its root hash (which is 32 bytes) into an `OP_RETURN` Bitcoin transaction to broadcast to the network. Sampled over 1,000 runs, the average time to generate this transaction — in the standard case of one input and two outputs, one for `OP_RETURN` and one for the authority's change — was 0.03 s (σ =0.007 s). The average total time to run `commit` was thus 5.93 s, as seen in Table 6.3, and it resulted in 235 bytes (the size of the transaction) being broadcast to the network.

Next, to run `prove_incl`, the authority proceeds in two phases: first constructing the Merkle proof for their transaction within the block where it eventually appears, and next constructing the Merkle proof for each statement represented in a transaction. The time for the first phase, averaged over 1M runs and for a block with 6,023 transactions (our upper bound from Section 6.6.2), was 8.5 µs. This is denoted "one-time" in Table 6.3 as it is done only once per batch. The time for the second phase, averaged over 1M runs, was 12 µs for each individual statement (thus denoted "per statement" in Table 6.3). Generating inclusion proofs for all the statements in the batch would thus take around 12 s. In terms of bandwidth and storage, the block may be up to 1 MB in size. In terms of the memory costs, the size of the Merkle tree for 1M leafs in memory is 649MB.

Additionally, in order to ensure that its transaction makes it into a block quickly, the authority may want to pay a fee. The current recommended rate is 16 satoshis/byte (`https://bitcoinfees.21.co/`), so for a 235-byte transaction the authority can expect to pay 3,760 satoshis. As of December 2019, this is roughly 0.28 USD.

Auditor overheads

For the auditor, we considered two costs: the initial cost to retrieve the necessary header data (`sync`), and the cost to verify an inclusion proof (`check_incl`). We do not provide performance benchmarks for the Auditor.`get_arch_state` call, as this is a simple web request that returns a single 32-byte hash.

To run `sync`, auditors use the Bitcoin SPV protocol to download and verify the headers of each block, which are 80 bytes each. As of this writing on December 2019, there are 608,000 valid mined blocks, which equates to 48.6 MB of block headers. Once downloaded, however, the auditor needs to keep only the 32-byte block hash, so only 19.5 MB of data needs to be stored on disk. Going forward, the Bitcoin network generates approximately 144 blocks per day, so the amount of downloaded data will increase by 11.5 kB daily, and the amount of stored data by 4.6 kB daily.

To verify the validity of the block headers in the chain, the client must perform one SHA-256 hash per block header; on average, it took us on average 116 seconds (over 5 runs) for the Python SPV client to download and verify all the block headers from the network. This initial bootstrapping process needs to be performed only once per auditor.

To run `check_incl`, we again use our upper bound from Section 6.6.2 and assume every block contains 6,023 transactions. This means the inclusion proof contains: (1) an 80-byte block header; (2) the raw transaction data, which is 235 bytes; (3) a Merkle proof for the transaction, which consists of $\log(6,023) - 1$ 32-byte hashes (the root hash is already provided in the block header); and (4) a Merkle proof for the statement, which consists of $\log(1,000,000) - 2$ 32-byte hashes (the root hash is already provided in the transaction data, and the auditor computes the statement hash itself). The total bandwidth cost is therefore around 1275 bytes. Averaged over 1M runs, the time for the auditor to verify the inclusion proof was 224 µs ($\sigma = 62.14$ µs).

To confirm that the time to run `check_incl` scales logarithmically with the number of statements in the batch, we also changed this number. The results are in

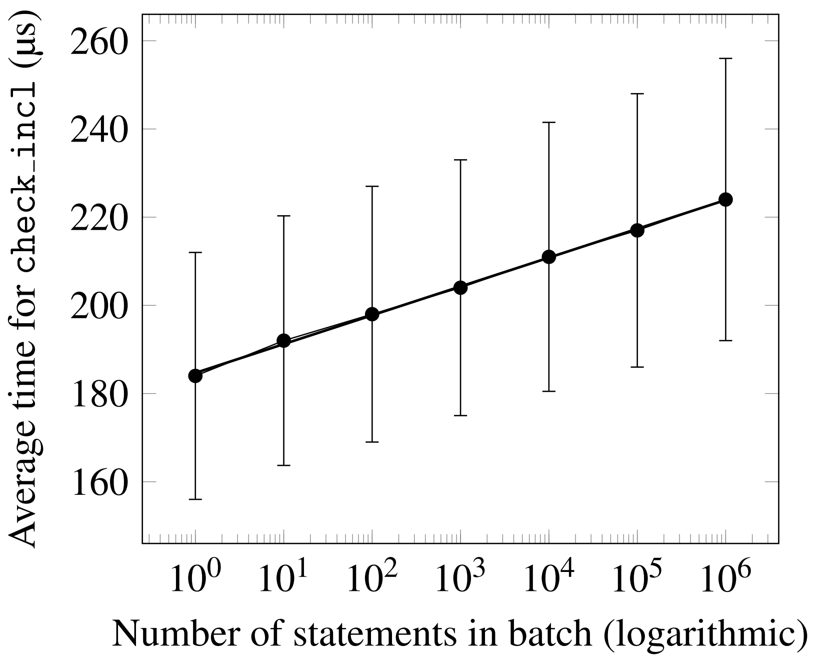


Figure 6.4: The time to verify an inclusion proof with varying numbers of statements in the batch, averaged over 100K runs.

Figure 6.4 (and do confirm logarithmic scaling).

Monitor overheads

Monitors must run a Bitcoin full node in order to get a complete uncensored view of the blockchain. As of December 2019, running a Bitcoin full node requires at least 245 GB of free disk space [98], increasing by up to 144 MB daily. It took us around three days to fully bootstrap a Bitcoin full node and verify all the blocks, although again this operation needs to be performed only once per monitor.

6.7 Use Case: Debian

To demonstrate how Contour can be used on a real system, we prototyped it for auditing software binaries in the Debian software repository. Our results show that Contour provides a way to add transparency to this repository without major changes to the existing infrastructure and with minimal overheads. It could be deployed on top of the Debian ecosystem today, without any participant who did not want to opt in having to change their behaviour.

We begin with an overview of how Debian currently works, and then go on to explain how existing actors in the ecosystem could play the roles necessary for Contour, along with the overheads.

6.7.1 Software distribution architecture

Debian is a popular Linux distribution used by over 32% of websites that run Linux.[6] Software packages are installed and updated on Debian machines using the `apt` command-line program. The Debian software repository contains `Release` files for various versions of Debian, which are updated every time any package in the repository is updated. Each `Release` file contains a checksum for a `Packages` file, which contains a list of available software packages and their associated checksums for integrity checking.

Software packages are downloaded as `.deb` archives which provide the compiled binaries and scripts required to install a package on a system. These files are hosted in directories on HTTP mirrors, of which hundreds exist around the world.[7]

To cryptographically authenticate software packages, Debian has a set of tools called `apt-secure`. Debian installations come with a built-in set of PGP keys [69] that are used as trusted keys for validating software packages. Alongside the `Release` files in the repository, there are `Release.gpg` files that contain PGP signatures of the `Release` files under trusted PGP keys.[8]

Through the single signature of a `Release` file, `apt` can validate that individual `.deb` packages were authorised by a trusted PGP key by checking that the checksums of packages are included in the `Packages` file whose checksum is included in the root `Release` file. This of course creates a central point of failure, as the owner of the signing key can serve individual users targeted `Release` files — for example, if coerced to do so by law enforcement — that link to malicious packages.

[6] `https://w3techs.com/technologies/details/os-linux/all/all`

[7] `https://www.debian.org/mirror/list`

[8] `https://wiki.debian.org/SecureApt`

6.7.2 Authority

In the case of Debian software distribution, the most natural operators for a Contour authority are the maintainers of the software repository. Specifically, the Contour authority would be the owner of the PGP key, as only this entity has the power to modify the software repository. Importantly, it is also possible for third parties to act as Contour authorities by proxy and commit binaries to the log on behalf of the maintainers of the Debian software repository. As committed binaries are transparent, the third party is not trusted any more than the maintainers of the Debian software repository would be, as any rogue additions to the log would still be detectable. This means it would be possible to deploy Contour today without any intervention or permission from the Debian project itself.

To initiate the system as an authority, all the existing software packages would first need to be committed; i.e., the authority would need to commit to the current state of the repository. To measure the overhead needed for this step, we extracted the software package metadata for all processor architectures and releases of Debian from the Debian FTP archive[9] over a one-week period from January 20-27 2017. At the beginning of this period there were 976,214 unique software binaries available for download from the Debian software repositories, constituting 1.7 TB of data, and by the end there were 980,469.

As discussed above, the Debian package metadata already contains a SHA-256 hash for every packages, so we needed only to build a Merkle tree from these hashes (rather than compute them ourselves first), to then commit on the blockchain. This took approximately 6 seconds (which is in line with our benchmarks in Table 6.3 for 1M statements).

Going forward, the authority must commit batches of new and updated binaries to the log. The Debian FTP archives are updated four times a day, which means four batches to commit to the log per day. Recall from Section 6.6.2 that committing one transaction to the blockchain currently costs roughly 0.28 USD in fees, so this would cost 1.12 USD per day (although, as mentioned in Section 6.6, Bitcoin prices

[9] `https://www.debian.org/mirror/ftpmirror`

are notoriously volatile). This is a relatively low price to pay for a system that costs over 91M USD to attack (Section 6.5.1).

As the archive was updated, we kept track of the package hashes being added and created a new batch for each update. The average batch size was 1,040 packages, and the average time to build a Merkle tree for the batch was 0.0052 seconds.

Finally, the proof of inclusion of each software package would need to be stored alongside each software package (`.deb`) file as metadata to be downloaded by Debian machines. At 980K software packages, this would require a maximum of 1.3 kB of extra storage per package, or 1.3 GB of extra storage to store the proofs of inclusion for all packages. Given the current storage requirements of (at least) 1.7 TB, this is only a 0.07% overhead.

6.7.3 Auditors

On the end-user side, the `apt` program would need to be modified to integrate the Auditor.`check_incl` and Auditor.`sync` calls, as implemented and analyzed in Section 6.6. This would ensure that downloaded packages are in the log before being installed.

In terms of overhead for end-user Debian machines, as discussed above this would require an extra 1.3 kB of bandwidth per package downloaded or updated. Given that the average package size is 1337 kB, the average overhead is 0.1% per package. We stress that this is a bandwidth requirement only, as once the proofs of inclusion are verified they do not need to be stored on the client's machine.

On a freshly installed Debian 8.8 system there are 520 packages installed by default, with a total `.deb` archive size of 190 MB. Verifying that each of these are in the log would require an extra 698.1 kB of bandwidth, and would take under two minutes.

6.7.4 Monitors

Debian's reproducible builds project allows any interested parties to verify that binaries published in the software repositories are compiled from a given source code.[10]

[10]`https://wiki.debian.org/ReproducibleBuilds`

There are no specific parties assigned to the role of monitoring builds to see if they can be built from the source code. Similarly in Contour, any parties vested in the security of Debian may act as a monitor. Aside from end users, we anticipate that large organizations supplying critical infrastructure using Debian, national CERTs, and NGOs such as the Electronic Frontier Foundation would have an interest in monitoring the log.

Generally, any party that wants extra guarantees about the software updates they are installing — e.g., in order to be sure that the updates that have been pushed to their machines are the same as those that have been pushed to other machines — should run a monitor. For example, if a party running Debian receives $\mathtt{update}_1$ and $\mathtt{update}_3$ on their machine for some software package, but the log contains $\mathtt{update}_1$, $\mathtt{update}_2$, and $\mathtt{update}_3$, then this raises a red flag as to why they did not receive $\mathtt{update}_2$. In particular, $\mathtt{update}_2$ may be a malicious update targeted to specific machines, and the party can check to see if the contents of $\mathtt{update}_2$ have been made available by the authority. If they have not, then the authority is considered to be misbehaving. prevent auditors from accepting the update altogether.

6.7.5 Summary

In summary, Contour could be deployed on top of the existing system for Debian software distribution with minimal changes to the existing infrastructure. Costs are minimal, with only a 0.07% storage overhead required for the authority, and a 0.1% bandwidth overhead for the end user. The computational costs for these users are minimal as well.

One distinguishing feature of Contour is that no existing parties in the Debian infrastructure are required to participate if they do not want to, and as discussed earlier the security assumptions of the system would remain the same even if a third party acted as an authority. This places Contour in contrast to existing proposals for transparency, as they require the initial setup of some Sybil-free set of nodes. In contexts such as the distribution of Debian software packages, this assumption — and the security implications if it is violated — presents a significant obstacle to deployability, and avoiding this obstacle was one of our main goals in designing

Contour.

6.8 Discussion and Extensions

Selective disclosure. When releasing software updates that patch critical security vulnerabilities, some software vendors may prefer not to reveal to potential attackers that, in the window of time in which a commitment has not yet been included in the blockchain, they can take advantage of victims with this vulnerable software installed. In such a case, Contour accounts for this by allowing the authority to commit to a batch of binaries visibly on the blockchain, but delay the publication of the binaries themselves until the commitment is sufficiently deep in the blockchain.

Generalized transparency. Although we have designed Contour for the specific application of binary transparency, the system is general enough to be applied to other applications requiring transparency. It can even be applied to the setting of certificate transparency, using CAs as authorities.

6.9 Conclusion

We have proposed Contour, a system that provides proactive transparency, scales logarithmically for auditors, and does not require the initial coordination of a Sybil-free set of nodes. To the best of our knowledge, it is also the first system for providing binary transparency.

We have demonstrated that, even for attackers that are capable of performing (for free) persistent man-in-the-middle attacks and are targeted a single device, compromising the integrity of the system requires roughly 65M USD in energy and hardware costs. We also saw that Contour could be applied today to the Debian software repository with relatively minimal changes and overhead to existing infrastructure, with the main extra cost being the storage requirements of archival nodes that mirror the repository data. The overheads for end users, in contrast, are quite minimal, with the proof of inclusion for a binary within a batch of size 1M being only 1.3 kB and taking only 224 µs to verify.

Chapter 7

Conclusion

Amen.

The Bible

In this thesis, we designed and evaluated techniques to build scalable and secure decentralised on-chain protocols.

In Chapter 3 we introduced Chainspace, a sharded blockchain protocol that splits the state of the ledger into multiple chains so that blocks can be produced in parallel, thereby increasing on-chain transaction throughput. We introduced S-BAC, an atomic cross-shard transaction protocol, and showed that the overall system scales linearly with the number of shards. Our modest test-bed of 60 nodes achieved 350 transactions per second, compared to Bitcoin's 7 transactions per second over 5,000+ nodes.

We then tackled the problem of scaling block validation in Chapter 4, so that under-resourced nodes can efficiently make use of the public verifiability properties of blockchain systems with high on-chain throughput. We proposed a system of fraud proofs, which allows light clients to receive compact proofs that blocks contain invalid state transitions, instead of directly replaying the chain. Additionally, we also introduced the concept of data availability proofs, a technique to enable light clients to efficiently verify that block data was published, so that the data necessary to generate fraud proofs is available. We showed that with modest parameters, only 0.4% of block data needs to be downloaded to detect that a block is unavailable with greater than 99% probability, per client.

In Chapter 5 we used our new data availability proofs primitive to design a new layer one architecture for blockchains. In this architecture, the chain is solely used as 'verifiable log' used for transaction ordering and data availability, and all execution of transactions applications is performed client-side. We showed that this has several advantages, including the fact users of one application do not need to download or verify the state of other unrelated applications.

Finally, in Chapter 6 we designed an efficient verifiable log application that does not require any transaction execution, relying on Bitcoin's layer one network for economic anti-equivocation security. We proposed the application of software binary transparency, an application that is uniquely suited for Bitcoin due to the need for a proactive transparency mechanism. We showed that by using the Bitcoin chain for economic anti-equivocation guarantees, a malicious actor would require 65M USD in energy and hardware costs to target a device with a targeted software update without detection.

7.1 Future Research Directions

In this section, we propose some topics that may be interesting directions of future research, continuing the research done in this thesis.

Validity Proofs

In Chapter 4, we used fraud proofs for (*i*) detecting invalid state transitions in blocks and (*ii*) detecting incorrectly generated erasure coded Merkle roots for data availability proofs.

One of the disadvantages of relying on fraud proofs in such protocols is (*i*) fraud proofs cause finality delays, as the user must wait for a period in case there is a fraud proof and (*ii*) it requires a synchronous gossiping network. Instead of using fraud proofs, protocols can use validity proofs to efficiently prove that block data is valid, so that no such waiting period is necessary.

There have been advances in succinct proofs of computation, including zk-SNARKs [24] and more recently zk-STARKs [23], which allow a prover to prove that $f(x,W) = y$ for some provided x and y, where even if the witness W is very large in size and the computation f takes a very long time to compute, the proof itself has only logarithmic or constant size and takes logarithmic or constant time to verify.

Instead of relying on fraud proofs of incorrectly generated erasure codes, we can require block headers to come with such a proof to show that they are correctly erasure coded, removing the need for fraud proofs. Also note that the only significant advantage of the 2D Reed Solomon scheme over the 1D scheme is smaller fraud proofs, so if succinct proofs are used switching back to 1D may be optimal (constructing a legitimate erasure code takes only $O(n \log(n))$ computation time for n shares if Fast Fourier Transforms are used [97, 127]). Similarly, we can also require block headers to come with such a proof to show that the all the state transitions in the block are valid.

Layer One Censorship Resistance

Current design paradigms for consensus mechanisms cannot guarantee liveness in the presence of a dishonest majority of consensus participants. In particular, this creates the possibility of transaction censorship, as block producers may refuse to include certain transactions in blocks. Although this is seen as a liveness failure from the perspective of a layer one chain, this can result in safety failures for layer two sidechain protocols that rely on the layer one chain.

For example, some designs for optimistic rollup sidechains (see Section 5.2.4) rely on fraud proofs of invalid state transitions being processed by a smart contract on the main chain, in the event that an invalid sidechain block is created [1]. This fraud proof must be included in the chain within a defined challenge period, after which sidechain blocks are considered final. Therefore a dishonest consensus majority on the main chain can censor the inclusion of this fraud proof within the chain, thus causing a safety failure in the main chain.

An important topic of future research is therefore designing layer one systems that are censorship resistant in the presence of a dishonest majority of consensus participants. Two potential research directions for this include:

- Designing a chain for smart contracts where all transactions use privacy-preserving zero-knowledge proofs such as zk-SNARKs [24], in such a manner so that all transactions are indistinguishable from each other. If all transactions are indistinguishable from each other, then block producers cannot censors transactions based on their content. Blockchains such as Zcash [77] use zk-SNARKs to enable anonymous transactions, however anonymous transactions are optional and only payments are supported.

- Designing a chain with a fork-choice rule or block validity rule that prevents censorship of transactions deemed to be important (such as state transition fraud proofs for sidechains). One proposed design for this [5] introduces a block validity rule where blocks that do not contain important transactions known by the transaction memory pool of the network at a certain point in time, are invalid. This requires a weak subjectivity assumption, which means that if a node goes offline for too long, they must ask another trusted node for the correct fork of the chain. This is because the offline node does not know which transactions were in the network's memory pool at the time when it was offline.

More Efficient Data Availability Proofs

The 2D data availability proof scheme in Section 4.5 requires clients to download $2\sqrt{n}$ Merkle roots for a n-sized block. Additionally, downloading a fraud proof of an incorrectly generated erasure code in the 2D scheme requires downloading a $\sqrt{n}$-sized row or column. It is worth exploring alternative designs with lower overheads.

One such design that is based on this work is SPAR [156], which proposes a Merkle tree where the intermediate nodes at each level are encoded using an LDPC code [68]. Compared to our 2D Reed-Solomon scheme, only one Merkle root needs to be downloaded per block, and the fraud proof size is $O(\log(n))$. However, a greater number of samples are required to achieve the same level of data availability confidence. With similar parameters, SPAR requires 35 samples to achieve 99% confidence, whereas the 2D scheme requires only 17.

An important topic of future work would therefore be to design data availability schemes that are more efficient in terms of sampling cost, fraud proof size and overhead costs, while minimising any trade-offs in these costs.

Fraud Proofs and Sharding

In Chapter 4 we proposed a generic system of fraud proofs for invalid state transitions. However, we did not explore in much detail how such a system could be applied within a sharding protocol such as Chainspace in Chapter 3. We leave this to future work, and provide some discussion here.

The Chainspace sharding system in Chapter 3 was shown to provide linear scalability under a threat model where all shards are assumed to be honest. However under a threat model where shards are not assumed to be honest, fraud proofs would be required to allow each shard to detect invalid blocks from other shards. This would require each shard to run a light client for every shard, and download the block headers for each block in every shard. This requires an overhead of $O(s)$ where s is the number of shards in the system, per block. Therefore, there is a limit to the number of shards that the system may have, due to the bookkeeping of other shards required by each shard. Exploring such limits, and whether they can be overcome, is a topic of future research.

Designs for real-world sharding systems such as Ethereum 2.0 have similar limitations. In Ethereum 2.0 there is a 'beacon chain' that keeps track of all the block headers for all shards. The beacon chain can be thought of as a shard in its own right. Therefore if the computational capacity of each node in the network increases by n, then the beacon chain can keep track of n times more shards, and each shard can process n times more transactions. Therefore the transaction throughput of the system increases by n^2 – this is known as 'quadratic sharding' [137].

The state transition fraud proof system proposed in Chapter 4 was designed in the context of a single-chain system. When considering a sharded system, the cross-shard communication protocol must be taken into account, in case that any dishonest shard does not follow the protocol correctly. In the case of S-BAC, a shard may violate the protocol by committing a transaction that has been rejected by an input shard, or aborting a transaction that has been accepted by all shards. To prevent this, the transcript of the S-BAC protocol written into each shard's blockchain could also include a Merkle inclusion proof for each cross-shard message, justifying each shard's actions. These Merkle proofs are similar to the proofs of acceptance and rejection in OmniLedger [87]. This transcript can be used as an audit log, such that if an invalid transcript is published on a shard, this can be used as a fraud proof.

7.2 Closing Thoughts

At the start of this thesis, we posed the problem of scaling the base layer of blockchains, while at the same time not trading off the security and decentralisation properties that give blockchains their purpose. As was mentioned in Chapter 1, we were specifically concerned about the ability of end-users to verify (directly or indirectly) that the chain is valid.

The scaling solutions we proposed introduce new limitations and security assumptions compared to existing blockchains such as Bitcoin [109]. Therefore, it can be argued that some security was traded to achieve scalability. Whether the new limitations and assumptions are acceptable is often a subjective matter that depends on the real-world context where the systems are being deployed.

The work on sharding in Chapter 3 increases the resources required by end-users to ensure the validity of the state of the system. Fraud and data availability proofs in Chapter 4 alleviate this, but this introduces assumptions about the minimum number of light clients in the network. We have argued in Section 4.5 that in practice this assumption is very likely to be met. Additionally, fraud proofs increase transaction latency as clients must wait to see if they receive a fraud proof before accepting new blocks.

The work on LazyLedger in Chapter 5 makes use of data availability proofs and thus relies on the same assumptions. On the other hand, the work on Contour in Chapter 6 does not have these limitations as it makes use of Merkle roots pegged on the Bitcoin blockchain for security, but this is only suitable for applications that do not require any execution.

To conclude, we have found that it is possible to increase on-chain throughput with better security and decentralisation trade-offs than simply increasing block size, but ultimately a trade-off is still required. It is time for us to consider what trade-offs are acceptable and worth making to bring cryptocurrencies to the levels of scalability required for mainstream adoption.

Bibliography

[1] Adler, J.: Minimal viable merged consensus (2019), https://ethresear.ch/t/minimal-viable-merged-consensus/5617

[2] Al-Bassam, M.: SCPKI: A smart contract-based PKI and identity system. In: ACM Workshop on Blockchain, Cryptocurrencies and Contracts. pp. 35–40. BCC '17, ACM, New York, NY, USA (2017). doi: 10.1145/3055518.3055530, https://doi.org/10.1145/3055518.3055530

[3] Al-Bassam, M.: New threat models in the face of British intelligence and the Five Eyes' new end-to-end encryption interception strategy (2018), https://www.benthamsgaze.org/2018/12/06/new-threat-models-in-the-face-of-british-intelligence-and-the-five-eyes-new-end-to-end-encryption-interception-strategy/

[4] Al-Bassam, M.: LazyLedger: A distributed data availability ledger with client-side smart contracts. CoRR **abs/1905.09274** (2019), http://arxiv.org/abs/1905.09274

[5] Al-Bassam, M.: Simple censorship-resistance for on-chain fraud proofs via weak subjectivity (2020), https://ethresear.ch/t/simple-censorship-resistance-for-on-chain-fraud-proofs-via-weak-subjectivity/6523

[6] Al-Bassam, M., Meiklejohn, S.: Contour: A practical system for binary transparency. In: García-Alfaro, J., Herrera-Joancomartí, J., Livraga, G., Rios, R. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11025, pp. 94–110. Springer (2018). doi: 10.1007/978-3-030-00305-0_8, `https://doi.org/10.1007/978-3-030-00305-0_8`

[7] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., Danezis, G.: 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. The Internet Society (2018), `http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-2_Al-Bassam_paper.pdf`

[8] Al-Bassam, M., Sonnino, A., Buterin, V.: Fraud proofs: Maximising light client security and scaling blockchains with dishonest majorities. CoRR **abs/1809.09044** (2018), `http://arxiv.org/abs/1809.09044`

[9] Al-Bassam, M., Sonnino, A., Król, M., Psaras, I.: Airtnt: Fair exchange payment for outsourced secure enclave computations. CoRR **abs/1805.06411** (2018), `http://arxiv.org/abs/1805.06411`

[10] Amsden, Z., Arora, R., Bano, S., Baudet, M., Blackshear, S., Bothra, A., Cabrera, G., Catalini, C., Chalkias, K., Cheng, E., Ching, A., Chursin, A., Danezis, G., Giacomo, G.D., Dill, D.L., Ding, H., Doudchenko, N., Gao, V., Gao, Z., Garillot, F., Gorven, M., Hayes, P., Hou, J.M., Hu, Y., Hurley, K., Lewi, K., Li, C., Li, Z., Malkhi, D., Margulis, S., Maurer, B., Mohassel, P., de Naurois, L., Nikolaenko, V., Nowacki, T., Orlov, O., Perelman, D., Pott, A., Proctor, B., Qadeer, S., Rain, Russi, D., Schwab, B., Sezer, S., Sonnino, A., Venter, H., Wei, L., Werner-

felt, N., Williams, B., Wu, Q., Yan, X., Zakian, T., Zhou, R.: The Libra blockchain (2019), `https://developers.libra.org/docs/assets/papers/the-libra-blockchain.pdf`

[11] Antonopoulos, A.M.: Mastering Bitcoin: Unlocking Digital Crypto-Currencies. O'Reilly Media, Inc., 1st edn. (2014)

[12] Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Kissner, L., Peterson, Z.N.J., Song, D.X.: Provable data possession at untrusted stores. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 598–609. ACM (2007). doi: 10.1145/1315245.1315318, `https://doi.org/10.1145/1315245.1315318`

[13] Ateniese, G., Pietro, R.D., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. In: Levi, A., Liu, P., Molva, R. (eds.) 4th International ICST Conference on Security and Privacy in Communication Networks, SECURECOMM 2008, Istanbul, Turkey, September 22-25, 2008. p. 9. ACM (2008). doi: 10.1145/1460877.1460889, `https://doi.org/10.1145/1460877.1460889`

[14] Avarikioti, G., Kokoris-Kogias, E., Wattenhofer, R.: Divide and scale: Formalization of distributed ledger sharding protocols. CoRR **abs/1910.10434** (2019), `http://arxiv.org/abs/1910.10434`

[15] Awerbuch, B., Scheideler, C.: Towards a scalable and robust DHT. Theory Comput. Syst. **45**(2), 234–260 (2009). doi: 10.1007/s00224-008-9099-9, `https://doi.org/10.1007/s00224-008-9099-9`

[16] Azouvi, S., Al-Bassam, M., Meiklejohn, S.: Who am I? secure identity registration on distributed ledgers. In: García-Alfaro, J., Navarro-Arribas, G., Hartenstein, H., Herrera-Joancomartí, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2017 International

Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10436, pp. 373–389. Springer (2017). doi: 10.1007/978-3-319-67816-0_21, `https://doi.org/10.1007/978-3-319-67816-0_21`

[17] Azouvi, S., McCorry, P., Meiklejohn, S.: Betting on blockchain consensus with fantomette. CoRR **abs/1805.06786** (2018), `http://arxiv.org/abs/1805.06786`

[18] Back, A.: A partial hash collision based postage scheme (1997), `http://www.hashcash.org/papers/announce.txt`

[19] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., Wuille, P.: Enabling blockchain innovations with pegged sidechains (2014), `https://blockstream.com/sidechains.pdf`

[20] Bano, S., Sonnino, A., Al-Bassam, M., Azouvi, S., McCorry, P., Meiklejohn, S., Danezis, G.: SoK: Consensus in the age of blockchains. In: Proceedings of the 1st ACM Conference on Advances in Financial Technologies, AFT 2019, Zurich, Switzerland, October 21-23, 2019. pp. 183–198. ACM (2019). doi: 10.1145/3318041.3355458, `https://doi.org/10.1145/3318041.3355458`

[21] Barnes, R.: Security/binary transparency - MozillaWiki (2017), `https://wiki.mozilla.org/index.php?title=Security/Binary_Transparency&oldid=1163743`

[22] Basin, D.A., Cremers, C.J.F., Kim, T.H., Perrig, A., Sasse, R., Szalachowski, P.: ARPKI: attack resilient public-key infrastructure. In: Ahn, G., Yung, M., Li, N. (eds.) Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014. pp. 382–393. ACM (2014). doi: 10.1145/2660267.2660298, `https://doi.org/10.1145/2660267.2660298`

[23] Ben-Sasson, E., Bentov, I., Horesh, Y., Riabzev, M.: Scalable, transparent, and post-quantum secure computational integrity. IACR Cryptology ePrint Archive **2018**, 46 (2018)

[24] Ben-Sasson, E., Chiesa, A., Tromer, E., Virza, M.: Succinct non-interactive zero knowledge for a von Neumann architecture. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 781–796. USENIX Association, San Diego, CA (2014), `https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson`

[25] Benet, J.: IPFS - content addressed, versioned, P2P file system. CoRR **abs/1407.3561** (2014), `http://arxiv.org/abs/1407.3561`

[26] Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)

[27] Bessani, A.N., Sousa, J., Alchieri, E.A.P.: State machine replication for the masses with BFT-SMART. In: 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014. pp. 355–362. IEEE Computer Society (2014). doi: 10.1109/DSN.2014.43, `https://doi.org/10.1109/DSN.2014.43`

[28] Bhatnagar, N., Miller, E.L.: Designing a secure reliable file system for sensor networks. In: Henson, V. (ed.) Proceedings of the 2007 ACM Workshop On Storage Security And Survivability, StorageSS 2007, Alexandria, VA, USA, October 29, 2007. pp. 19–24. ACM (2007). doi: 10.1145/1314313.1314319, `https://doi.org/10.1145/1314313.1314319`

[29] Blummer, T., Bohan, S., Bowman, M., Cachin, C., Gaski, N., George, N., Graham, G., Hardman, D., Jagadeesan, R., Keith, T., Khasanshyn, R., Krishna, M., Kuhrt, T., Hors, A.L., Levi, J., Liberman, S., Mendez, E., Middleton, D., Montgomery, H., O'Prey, D., Reed, D., Teis, S., Voell, D., Wallace, G., Yang, B.: An introduction to Hyperledger (2018), `https:`

//www.hyperledger.org/wp-content/uploads/2018/07/HL_Whitepaper_IntroductiontoHyperledger.pdf

[30] Bobbio, J.: Reproducible builds for Debian (2014), https://archive.fosdem.org/2014/schedule/event/reproducibledebian/

[31] Boneh, D., Bünz, B., Fisch, B.: Batching techniques for accumulators with applications to IOPs and stateless blockchains. In: Boldyreva, A., Micciancio, D. (eds.) Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11692, pp. 561–586. Springer (2019). doi: 10.1007/978-3-030-26948-7_20, https://doi.org/10.1007/978-3-030-26948-7_20

[32] Bonneau, J.: Ethiks: Using ethereum to audit a CONIKS key transparency log. In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9604, pp. 95–105. Springer (2016). doi: 10.1007/978-3-662-53357-4_7, https://doi.org/10.1007/978-3-662-53357-4_7

[33] Bowers, K.D., Juels, A., Oprea, A.: Proofs of retrievability: theory and implementation. In: Sion, R., Song, D. (eds.) Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009, Chicago, IL, USA, November 13, 2009. pp. 43–54. ACM (2009). doi: 10.1145/1655008.1655015, https://doi.org/10.1145/1655008.1655015

[34] Boyd, S.P., Ghosh, A., Prabhakar, B., Shah, D.: Randomized gossip algorithms. IEEE Trans. Inf. Theory **52**(6), 2508–2530 (2006). doi: 10.1109/TIT.2006.874516, https://doi.org/10.1109/TIT.2006.874516

[35] Buchman, E., Kwon, J., Milosevic, Z.: The latest gossip on BFT consensus. CoRR **abs/1807.04938** (2018), `http://arxiv.org/abs/1807.04938`

[36] Buterin, V.: Ethereum: The ultimate smart contract and decentralized application platform (white paper) (2013), `http://web.archive.org/web/20131228111141/http://vbuterin.com/ethereum.html`

[37] Buterin, V.: Cross-shard contract yanking (2018), `https://ethresear.ch/t/cross-shard-contract-yanking/1450`

[38] Buterin, V.: The dawn of hybrid layer 2 protocols (2019), `https://vitalik.ca/general/2019/08/28/hybrid_layer_2.html`

[39] Buterin, V.: Ethereum 2.0 phase 2 – data availability proofs (2019), `https://github.com/ethereum/eth2.0-specs/blob/e3549dec9ba1cf23807db447de8441402f546641/specs/core/2_data-availability-proofs.md`

[40] Buterin, V.: A nearly-trivial-on-zero-inputs 32-bytes-long collision-resistant hash function (2019), `https://ethresear.ch/t/a-nearly-trivial-on-zero-inputs-32-bytes-long-collision-resistant-hash-function/5511`

[41] Buterin, V., Griffith, V.: Casper the friendly finality gadget. CoRR **abs/1710.09437** (2017), `http://arxiv.org/abs/1710.09437`

[42] Cachin, C., et al.: Architecture of the Hyperledger blockchain Fabric. In: Workshop on distributed cryptocurrencies and consensus ledgers. vol. 310 (2016), `https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf`

[43] Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Seltzer, M.I., Leach, P.J. (eds.) Proceedings of the Third USENIX Symposium on Operat-

ing Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999. pp. 173–186. USENIX Association (1999), `https://dl.acm.org/citation.cfm?id=296824`

[44] Chase, M., Meiklejohn, S.: Transparency overlays and applications. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 168–179. ACM (2016). doi: 10.1145/2976749.2978404, `https://doi.org/10.1145/2976749.2978404`

[45] Chaum, D.: Untraceable electronic mail, return addresses, and digital pseudonyms. Commun. ACM **24**(2), 84–88 (1981). doi: 10.1145/358549.358563, `http://doi.acm.org/10.1145/358549.358563`

[46] Chuat, L., Szalachowski, P., Perrig, A., Laurie, B., Messeri, E.: Efficient gossip protocols for verifying the consistency of certificate logs. In: 2015 IEEE Conference on Communications and Network Security, CNS 2015, Florence, Italy, September 28-30, 2015. pp. 415–423. IEEE (2015). doi: 10.1109/CNS.2015.7346853, `https://doi.org/10.1109/CNS.2015.7346853`

[47] Cohen, B.: The BitTorrent protocol specification (2008), `https://www.bittorrent.org/beps/bep_0003.html`

[48] Cox, R., Valsorda, F.: Proposal: Secure the public go module ecosystem (2019), `https://golang.org/design/25530-sumdb`

[49] Croman, K., Decker, C., Eyal, I., Gencer, A.E., Juels, A., Kosba, A.E., Miller, A., Saxena, P., Shi, E., Sirer, E.G., Song, D., Wattenhofer, R.: On scaling decentralized blockchains - (A position paper). In: Clark, J., Meiklejohn, S., Ryan, P.Y.A., Wallach, D.S., Brenner, M., Rohloff, K. (eds.) Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9604, pp.

106–125. Springer (2016). doi: 10.1007/978-3-662-53357-4_8, https://doi.org/10.1007/978-3-662-53357-4_8

[50] Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper-evident logging. In: Proceedings of the 18th Conference on USENIX Security Symposium. pp. 317–334. SSYM'09, USENIX Association, Berkeley, CA, USA (2009), http://dl.acm.org/citation.cfm?id=1855768.1855788

[51] Dahlberg, R., Pulls, T., Peeters, R.: Efficient sparse merkle trees - caching strategies and secure (non-)membership proofs. In: Brumley, B.B., Röning, J. (eds.) Secure IT Systems - 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016, Proceedings. Lecture Notes in Computer Science, vol. 10014, pp. 199–215 (2016). doi: 10.1007/978-3-319-47560-8_13, https://doi.org/10.1007/978-3-319-47560-8_13

[52] Danezis, G., Hrycyszyn, D.: Blockmania: from block dags to consensus. CoRR **abs/1809.01620** (2018), http://arxiv.org/abs/1809.01620

[53] Danezis, G., Meiklejohn, S.: Centrally banked cryptocurrencies. In: 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016. The Internet Society (2016), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/centrally-banked-cryptocurrencies.pdf

[54] Deshpande, S.: Distributed Systems. Technical Publications (2009)

[55] Dijkstra, E.W.: Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569– (Sep 1965). doi: 10.1145/365559.365617, http://doi.acm.org/10.1145/365559.365617

[56] Douceur, J.R.: The sybil attack. In: Druschel, P., Kaashoek, M.F., Rowstron, A.I.T. (eds.) Peer-to-Peer Systems, First International Workshop,

IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2429, pp. 251–260. Springer (2002). doi: 10.1007/3-540-45748-8_24, `https://doi.org/10.1007/3-540-45748-8_24`

[57] Dudáček, L., Veřtát, I.: Multidimensional parity check codes with short block lengths. In: Telecommunications Forum (TELFOR), 2016 24th. pp. 1–4. IEEE (2016)

[58] Dwork, C., Lynch, N.A., Stockmeyer, L.J.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (1988). doi: 10.1145/42282.42283, `http://doi.acm.org/10.1145/42282.42283`

[59] Eijdenberg, A., Laurie, B., Cutter, A.: Verifiable Data Structures (2015), `https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf`

[60] Elias, P.: Error-free coding. Trans. of the IRE Professional Group on Information Theory (TIT) **4**, 29–37 (1954). doi: 10.1109/TIT.1954.1057464, `https://doi.org/10.1109/TIT.1954.1057464`

[61] Euler, L.: Solutio quarundam quaestionum difficiliorum in calculo probabilium. Opuscula Analytic **2**, 331–346 (1785)

[62] Ewasm contributors: Ewasm design overview and specification (2019), `https://github.com/ewasm/design/tree/38eeded28765f3e193e12881ea72a6ab807a3371`

[63] Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. In: Christin, N., Safavi-Naini, R. (eds.) Financial Cryptography and Data Security - 18th International Conference, FC 2014, Christ Church, Barbados, March 3-7, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8437, pp. 436–454. Springer (2014). doi: 10.1007/978-3-662-45472-5_28, `https://doi.org/10.1007/978-3-662-45472-5_28`

[64] Farivar, C.: Judge: Apple must help FBI unlock San Bernardino shooter's iPhone (2016), `https://arstechnica.com/tech-policy/2016/02/judge-apple-must-help-fbi-unlock-san-bernardino-shooters-iphone/`

[65] Ferrante, M., Saltalamacchia, M.: The coupon collector's problem. Materials matemàtics pp. 0001–35 (2014)

[66] Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985). doi: 10.1145/3149.214121, `https://doi.org/10.1145/3149.214121`

[67] Fromknecht, C., Velicanu, D., Yakoubov, S.: A decentralized public key infrastructure with identity retention. IACR Cryptology ePrint Archive, Report 2014/803 (2014), `http://eprint.iacr.org/2014/803.pdf`

[68] Gallager, R.G.: Low-density parity-check codes. IRE Trans. Inf. Theory **8**(1), 21–28 (1962). doi: 10.1109/TIT.1962.1057683, `https://doi.org/10.1109/TIT.1962.1057683`

[69] Garfinkel, S.: PGP: Pretty Good Privacy. O'Reilly & Associates (1994)

[70] Godbole, O.: Bitcoin looks south after strong rejection above $7,600 (2019), `https://www.coindesk.com/bitcoin-looks-south-after-strong-rejection-above-7600`

[71] Goodin, D.: Google warns of unauthorized TLS certificates trusted by almost all OSes (2015), `https://arstechnica.com/security/2015/03/google-warns-of-unauthorized-tls-certificates-trusted-by-almost-all-oses/`

[72] Gray, J., Lamport, L.: Consensus on transaction commit. ACM Trans. Database Syst. **31**(1), 133–160 (2006). doi: 10.1145/1132863.1132867, `https://doi.org/10.1145/1132863.1132867`

[73] Han, S., Liu, S., Chen, K., Gu, D.: Proofs of retrievability based on MRD codes. In: Huang, X., Zhou, J. (eds.) Information Security Practice and Experience - 10th International Conference, ISPEC 2014, Fuzhou, China, May 5-8, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8434, pp. 330–345. Springer (2014). doi: 10.1007/978-3-319-06320-1_25, `https://doi.org/10.1007/978-3-319-06320-1_25`

[74] Harmony Team: Harmony technical whitepaper version 2.0 (2019), `https://harmony.one/whitepaper.pdf`

[75] Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on Bitcoin's peer-to-peer network. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 129–144. USENIX Association (2015), `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman`

[76] Hicks, A., Mavroudis, V., Al-Bassam, M., Meiklejohn, S., Murdoch, S.J.: VAMS: verifiable auditing of access to confidential data. CoRR **abs/1805.04772** (2018), `http://arxiv.org/abs/1805.04772`

[77] Hopwood, D., Bowe, S., Hornby, T., Wilcox, N.: Zcash protocol specification (2020), `https://github.com/zcash/zips/raw/master/protocol/protocol.pdf`

[78] Juels, A., Jr., B.S.K.: Pors: proofs of retrievability for large files. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007. pp. 584–597. ACM (2007). doi: 10.1145/1315245.1315317, `https://doi.org/10.1145/1315245.1315317`

[79] Juels, A., Kelley, J., Tamassia, R., Triandopoulos, N.: Falcon codes: Fast, authenticated LT codes (or: Making rapid tornadoes unstoppable). In: Ray, I.,

Li, N., Kruegel, C. (eds.) Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 1032–1047. ACM (2015). doi: 10.1145/2810103.2813728, https://doi.org/10.1145/2810103.2813728

[80] Kalodner, H.A., Goldfeder, S., Chen, X., Weinberg, S.M., Felten, E.W.: Arbitrum: Scalable, private smart contracts. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1353–1370. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner

[81] Karlo, T.: Ending Bitcoin support (2018), https://stripe.com/blog/ending-Bitcoin-support

[82] Keoun, B.: Canaan's post-IPO stock plunge reveals sales slump, price war with Bitmain (2019), https://www.coindesk.com/canaans-post-ipo-stock-plunge-reveals-sales-slump-price-war-with-bitmain

[83] Kiayias, A., Russell, A., David, B., Oliynykov, R.: Ouroboros: A provably secure proof-of-stake blockchain protocol. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10401, pp. 357–388. Springer (2017). doi: 10.1007/978-3-319-63688-7_12, https://doi.org/10.1007/978-3-319-63688-7_12

[84] Kim, T.H., Huang, L., Perrig, A., Jackson, C., Gligor, V.D.: Accountable key infrastructure (AKI): a proposal for a public-key validation infrastructure. In: Schwabe, D., Almeida, V.A.F., Glaser, H., Baeza-Yates, R., Moon, S.B. (eds.) 22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013. pp. 679–690. International World Wide

Web Conferences Steering Committee / ACM (2013). doi: 10.1145/2488388.2488448, https://doi.org/10.1145/2488388.2488448

[85] King, S., Nadal, S.: PPCoin: Peer-to-peer crypto-currency with proof-of-stake (2012), https://www.peercoin.net/whitepapers/peercoin-paper.pdf

[86] Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., Ford, B.: Enhancing bitcoin security and performance with strong consistency via collective signing. In: 25th USENIX Security Symposium (USENIX Security 16). pp. 279–296. USENIX Association, Austin, TX (2016), https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kogias

[87] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., Ford, B.: 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. pp. 583–598. IEEE Computer Society (2018). doi: 10.1109/SP.2018.000-5, https://doi.org/10.1109/SP.2018.000-5

[88] Krohn, M.N., Freedman, M.J., Mazières, D.: On-the-fly verification of rateless erasure codes for efficient content distribution. In: 2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA. pp. 226–240. IEEE Computer Society (2004). doi: 10.1109/SECPRI.2004.1301326, https://doi.org/10.1109/SECPRI.2004.1301326

[89] Król, M., Sonnino, A., Al-Bassam, M., Tasiopoulos, A.G., Psaras, I.: Proof-of-prestige: A useful work reward system for unverifiable tasks. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2019, Seoul, Korea (South), May 14-17, 2019. pp. 293–301. IEEE (2019). doi:

10.1109/BLOC.2019.8751406, https://doi.org/10.1109/BLOC.2019.8751406

[90] Kulynych, B., Lueks, W., Isaakidis, M., Danezis, G., Troncoso, C.: ClaimChain: Improving the security and privacy of in-band key distribution for messaging. In: Lie, D., Mannan, M., Johnson, A. (eds.) Proceedings of the 2018 Workshop on Privacy in the Electronic Society, WPES@CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 86–103. ACM (2018). doi: 10.1145/3267323.3268947, https://doi.org/10.1145/3267323.3268947

[91] Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM Trans. Database Syst. **6**(2), 213–226 (1981). doi: 10.1145/319566.319567, https://doi.org/10.1145/319566.319567

[92] Lamport, L., Shostak, R.E., Pease, M.C.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. **4**(3), 382–401 (1982). doi: 10.1145/357172.357176, http://doi.acm.org/10.1145/357172.357176

[93] Laurie, B., Kasper, E.: Revocation transparency (2012), https://www.links.org/files/RevocationTransparency.pdf

[94] Laurie, B., Langley, A., Kasper, E.: Certificate transparency. Tech. rep. (2013), https://tools.ietf.org/html/rfc6962

[95] Lauwers, L., Willekens, M.: Five hundred years of bookkeeping: a portrait of luca pacioli. Tijdschrift voor Economie en Management **39**(3), 289–304 (1994)

[96] Leyden, J.: Inside 'operation black tulip': DigiNotar hack analysed (2011), https://www.theregister.co.uk/2011/09/06/diginotar_audit_damning_fail/

[97] Lin, S., Chung, W., Han, Y.S.: Novel polynomial basis and its application to reed-solomon erasure codes. In: 55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014. pp. 316–325. IEEE Computer Society (2014). doi: 10.1109/FOCS.2014.41, `https://doi.org/10.1109/FOCS.2014.41`

[98] Liu, S.: Bitcoin blockchain size 2010-2019, by quarter (2018), `https://www.statista.com/statistics/647523/worldwide-bitcoin-blockchain-size/`

[99] Luby, M.: LT codes. In: 43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings. p. 271. IEEE Computer Society (2002). doi: 10.1109/SFCS.2002.1181950, `https://doi.org/10.1109/SFCS.2002.1181950`

[100] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., Saxena, P.: A secure sharding protocol for open blockchains. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 17–30. ACM (2016). doi: 10.1145/2976749.2978389, `https://doi.org/10.1145/2976749.2978389`

[101] Malkhi, D.: Blockchain in the lens of BFT. USENIX Association, Boston, MA (Jul 2018), `https://www.usenix.org/conference/atc18/presentation/malkhi`

[102] Marshall, A.: Bitcoin scaling problem, explained (2017), `https://cointelegraph.com/explained/Bitcoin-scaling-problem-explained`

[103] Maxwell, G.: (2017), `https://botbot.me/freenode/bitcoin-wizards/2017-02-01/?msg=80297226&page=2`

[104] Maymounkov, P., Mazières, D.: Kademlia: A peer-to-peer information system based on the XOR metric. In: Druschel, P., Kaashoek, M.F., Rowstron, A.I.T. (eds.) Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers. Lecture Notes in Computer Science, vol. 2429, pp. 53–65. Springer (2002). doi: 10.1007/3-540-45748-8_5, `https://doi.org/10.1007/3-540-45748-8_5`

[105] Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: bringing key transparency to end users. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 383–398. USENIX Association (2015), `https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara`

[106] Merkle, R.C.: A digital signature based on a conventional encryption function. In: Pomerance, C. (ed.) Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings. Lecture Notes in Computer Science, vol. 293, pp. 369–378. Springer (1987). doi: 10.1007/3-540-48184-2_32, `https://doi.org/10.1007/3-540-48184-2_32`

[107] Miers, I., Garman, C., Green, M., Rubin, A.D.: Zerocoin: Anonymous distributed e-cash from bitcoin. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013. pp. 397–411. IEEE Computer Society (2013). doi: 10.1109/SP.2013.34, `https://doi.org/10.1109/SP.2013.34`

[108] Miller, A., Bentov, I., Bakshi, S., Kumaresan, R., McCorry, P.: Sprites and state channels: Payment networks that go faster than lightning. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security - 23rd Inter-

national Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11598, pp. 508–526. Springer (2019). doi: 10.1007/978-3-030-32101-7_30, https://doi.org/10.1007/978-3-030-32101-7_30

[109] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008), http://bitcoin.org/bitcoin.pdf

[110] Nakamoto, S.: Bitcoin genesis block (000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f) (2009)

[111] Nakashima, E., Miller, G., Tate, J.: U.s., Israel developed Flame computer virus to slow Iranian nuclear efforts, officials say (2012), https://www.washingtonpost.com/world/national-security/us-israel-developed-computer-virus-to-slow-iranian-nuclear-efforts-officials-say/2012/06/19/gJQA6xBPoV_story.html

[112] Nikitin, K., Kokoris-Kogias, E., Jovanovic, P., Gailly, N., Gasser, L., Khoffi, I., Cappos, J., Ford, B.: CHAINIAC: proactive software-update transparency via collectively signed skipchains and verified builds. In: Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017. pp. 1271–1287. USENIX Association (2017), https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/nikitin

[113] Nordberg, L., Gillmor, D., Ritter, T.: Gossiping in CT (2016), https://tools.ietf.org/html/draft-ietf-trans-gossip-03

[114] Ohta, K., Okamoto, T.: Multi-signature schemes secure against active insider attacks. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences **82**(1), 21–31 (1999)

[115] Orland, K.: Your Bitcoin is no good here—Steam stops accepting cryptocurrency (2017), `https://arstechnica.com/gaming/2017/12/steam-drops-Bitcoin-payment-option-citing-fees-and-volatility/`

[116] P4Titan: Slimcoin: A peer-to-peer crypto-currency with proof-of-burn (2014), `https://github.com/slimcoin-project/slimcoin-project.github.io/raw/master/whitepaperSLM.pdf`

[117] Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. In: Coron, J., Nielsen, J.B. (eds.) Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10211, pp. 643–673 (2017). doi: 10.1007/978-3-319-56614-6_22, `https://doi.org/10.1007/978-3-319-56614-6_22`

[118] Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM **27**(2), 228–234 (1980). doi: 10.1145/322186.322188, `http://doi.acm.org/10.1145/322186.322188`

[119] Pedersen, T.P.: Non-interactive and information-theoretic secure verifiable secret sharing. In: Feigenbaum, J. (ed.) Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings. Lecture Notes in Computer Science, vol. 576, pp. 129–140. Springer (1991). doi: 10.1007/3-540-46766-1_9, `https://doi.org/10.1007/3-540-46766-1_9`

[120] Perard, D., Lacan, J., Bachy, Y., Detchart, J.: Erasure code-based low storage blockchain node. In: IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), iThings/GreenCom/CPSCom/SmartData 2018, Halifax,

NS, Canada, July 30 - August 3, 2018. pp. 1622–1627. IEEE (2018). doi: 10.1109/Cybermatics_2018.2018.00271, https://doi.org/10.1109/Cybermatics_2018.2018.00271

[121] Peterson, W.W., Wesley, W., Weldon Jr Peterson, E., Weldon, E., Weldon, E.: Error-correcting codes. MIT press (1972)

[122] Poon, J., Buterin, V.: Plasma: Scalable autonomous smart contracts (2017), https://plasma.io/plasma-deprecated.pdf

[123] Poon, J., Dryja, T.: The Bitcoin Lightning network: Scalable off-chain instant payments (2016), https://lightning.network/lightning-network-paper.pdf

[124] Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. J. ACM **36**(2), 335–348 (1989). doi: 10.1145/62044.62050, https://doi.org/10.1145/62044.62050

[125] Ramachandran, A.: The life and death of Plasma (2020), https://medium.com/dragonfly-research/the-life-and-death-of-plasma-b72c6a59c5ad

[126] Ranvier, J.: Improving the ability of SPV clients to detect invalid chains (2017), https://gist.github.com/justusranvier/451616fa4697b5f25f60

[127] Reed, I.S., Scholtz, R.A., Truong, T., Welch, L.R.: The fast decoding of Reed-Solomon codes using Fermat theoretic transforms and continued fractions. IEEE Trans. Inf. Theory **24**(1), 100–106 (1978). doi: 10.1109/TIT.1978.1055816, https://doi.org/10.1109/TIT.1978.1055816

[128] Roio, D., Bria, F., Barritt, J., Hoepman, J.H., de Villiers, M., Samuel, P., Danezis, G., Demeyer, T., Bano, S., Sagarra, O.: DECODE whitepaper v1.0 (2018), https://decodeproject.github.io/whitepaper/

[129] Ryan, M.D.: Enhanced certificate transparency and end-to-end encrypted mail. In: 21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014. The Internet Society (2014), `https://www.ndss-symposium.org/ndss2014/enhanced-certificate-transparency-and-end-end-encrypted-mail`

[130] Saints, K., Heegard, C.: Algebraic-geometric codes and multidimensional cyclic codes: A unified theory and algorithms for decoding using grobner bases. IEEE Trans. Inf. Theor. **41**(6), 1733–1751 (Sep 2006). doi: 10.1109/18.476246, `https://doi.org/10.1109/18.476246`

[131] Saltzer, J.H., Reed, D.P., Clark, D.D.: End-to-end arguments in system design. ACM Trans. Comput. Syst. **2**(4), 277–288 (1984). doi: 10.1145/357401.357402, `https://doi.org/10.1145/357401.357402`

[132] Schwartz, D.: Hash tree - Ripple wiki (2013), `https://wiki.ripple.com/index.php?title=Hash_Tree&oldid=3120`

[133] Shacham, H., Waters, B.: Compact proofs of retrievability. J. Cryptology **26**(3), 442–483 (2013). doi: 10.1007/s00145-012-9129-2, `https://doi.org/10.1007/s00145-012-9129-2`

[134] Shea, J.M., Wong, T.F.: Multidimensional codes. Encyclopedia of Telecommunications (2003)

[135] Shen, B.Z., Tzeng, K.: Multidimensional extension of Reed-Solomon codes. In: Information Theory, 1998. Proceedings. 1998 IEEE International Symposium on. p. 54. IEEE (1998)

[136] Singh, A., Ngan, T., Druschel, P., Wallach, D.S.: Eclipse attacks on overlay networks: Threats and defenses. 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. IEEE (2006). doi: 10.1109/INFOCOM.2006.231, https://doi.org/10.1109/INFOCOM.2006.231

[137] Skidanov, A.: The authoritative guide to blockchain sharding, part 1 (2018), https://medium.com/nearprotocol/the-authoritative-guide-to-blockchain-sharding-part-1-1b53ed31e060

[138] Sonnino, A., Al-Bassam, M., Bano, S., Meiklejohn, S., Danezis, G.: Coconut: Threshold issuance selective disclosure credentials with applications to distributed ledgers. In: 26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019. The Internet Society (2019), https://www.ndss-symposium.org/ndss-paper/coconut-threshold-issuance-selective-disclosure-credentials-with-applications-to-distributed-ledgers/

[139] Sonnino, A., Bano, S., Al-Bassam, M., Danezis, G.: Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In: IEEE European Symposium on Security and Privacy 2020 (2020)

[140] Sousa, J., Bessani, A., Vukolic, M.: A byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In: 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018. pp. 51–58. IEEE Computer Society (2018). doi: 10.1109/DSN.2018.00018, https://doi.org/10.1109/DSN.2018.00018

[141] Sousa, J., Bessani, A.N.: From byzantine consensus to BFT state machine replication: A latency-optimal transformation. In: Constantinescu, C., Correia, M.P. (eds.) 2012 Ninth European Dependable Computing Conference, Sibiu, Romania, May 8-11, 2012. pp. 37–48. IEEE Computer Society (2012). doi: 10.1109/EDCC.2012.32, https://doi.org/10.1109/EDCC.2012.32

[142] Syta, E., Jovanovic, P., Kokoris-Kogias, E., Gailly, N., Gasser, L., Khoffi, I., Fischer, M.J., Ford, B.: Scalable bias-resistant distributed randomness. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 444–460. IEEE Computer Society (2017). doi: 10.1109/SP.2017.45, `https://doi.org/10.1109/SP.2017.45`

[143] Syta, E., Tamas, I., Visher, D., Wolinsky, D.I., Jovanovic, P., Gasser, L., Gailly, N., Khoffi, I., Ford, B.: Keeping authorities ”honest or bust” with decentralized witness cosigning. In: IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016. pp. 526–545. IEEE Computer Society (2016). doi: 10.1109/SP.2016.38, `https://doi.org/10.1109/SP.2016.38`

[144] Teutsch, J., Reitwießner, C.: A scalable verification solution for blockchains. arXiv preprint arXiv:1908.04756 (2019)

[145] Todd, P.: Fraud proofs (2016), `https://diyhpl.us/wiki/transcripts/mit-bitcoin-expo-2016/fraud-proofs-petertodd/`

[146] Tomescu, A., Devadas, S.: Catena: Efficient non-equivocation via bitcoin. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 393–409. IEEE Computer Society (2017). doi: 10.1109/SP.2017.19, `https://doi.org/10.1109/SP.2017.19`

[147] Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London mathematical society **2**(1), 230–265 (1937)

[148] Wang, C., Chow, S.S.M., Wang, Q., Ren, K., Lou, W.: Privacy-preserving public auditing for secure cloud storage. IEEE Trans. Computers **62**(2), 362–375 (2013). doi: 10.1109/TC.2011.245, `https://doi.org/10.1109/TC.2011.245`

[149] Wang, T.: Average retail electricity prices in the U.S. from 1990 to 2018 (2019), https://www.statista.com/statistics/183700/us-average-retail-electricity-price-since-1990/

[150] Wicker, S.B.: Reed-Solomon Codes and Their Applications. IEEE Press, Piscataway, NJ, USA (1994)

[151] Willett, J.R., Hidskes, M., Johnston, D., Gross, R., Schneider, M.: Omni protocol specification (2012), https://github.com/OmniLayer/spec

[152] Wong, J.I.: CryptoKitties is causing Ethereum network congestion (2017), https://qz.com/1145833/cryptokitties-is-causing-ethereum-network-congestion/

[153] Wood, G.: Ethereum: A secure decentralised generalised transaction ledger - Byzantium version, e94ebda (yellow paper) (2018), https://ethereum.github.io/yellowpaper/paper.pdf

[154] Wu, J., Jr., D.J.C.: New multilevel codes over gf(q). IEEE Trans. Inf. Theory **38**(3), 933–939 (1992). doi: 10.1109/18.135635, https://doi.org/10.1109/18.135635

[155] Yin, J., Martin, J., Venkataramani, A., Alvisi, L., Dahlin, M.: Separating agreement from execution for byzantine fault tolerant services. In: Scott, M.L., Peterson, L.L. (eds.) Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003. pp. 253–267. ACM (2003). doi: 10.1145/945445.945470, https://doi.org/10.1145/945445.945470

[156] Yu, M., Sahraei, S., Li, S., Avestimehr, S., Kannan, S., Viswanath, P.: Coded merkle tree: Solving data availability attacks in blockchains. In: Financial Cryptography and Data Security (2020)

[157] Zamani, M., Movahedi, M., Raykova, M.: Rapidchain: Scaling blockchain via full sharding. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.)

Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 931–948. ACM (2018). doi: 10.1145/3243734.3243853, https://doi.org/10.1145/3243734.3243853

[158] Zamyatin, A., Al-Bassam, M., Zindros, D., Kokoris-Kogias, E., Moreno-Sanchez, P., Kiayias, A., Knottenbelt, W.J.: SoK: communication across distributed ledgers. Cryptology ePrint Archive, Report 2019/1128 (2019), https://eprint.iacr.org/2019/1128

Appendix A

Fraud and Data Availability Proofs

A.1 Computation of index in Step 4 of VerifyCodecFraudProof

In Step 4 of VerifyCodecFraudProof in Section 4.5.7, index can be computed as follows:

- If $\mathsf{axis} = 0$ and $\mathsf{ax}_x = 0$, $\mathsf{index} = j * \mathsf{matrixWidth}_i + \mathsf{pos}_x$.
- If $\mathsf{axis} = 1$ and $\mathsf{ax}_x = 0$, $\mathsf{index} = \mathsf{pos}_x * \mathsf{matrixWidth}_i + j$.
- If $\mathsf{axis} = 1$ and $\mathsf{ax}_x = 1$, $\mathsf{index} = \frac{1}{2}\mathsf{dataLength_i} + j * \mathsf{matrixWidth}_i + \mathsf{pos}_x$.
- If $\mathsf{axis} = 0$ and $\mathsf{ax}_x = 1$, $\mathsf{index} = \frac{1}{2}\mathsf{dataLength_i} + \mathsf{pos}_x * \mathsf{matrixWidth}_i + j$.