Trifecta: the Blockchain TriLemma Solved

Trifecta Blockchain Team

October 2019

Abstract

We present Trifecta, a new blockchain which solves the Trilemma of decentralization, security and scalability: 1) it is fully permissionless; 2) it achieves full security, guaranteeing liveness and consistency against a fully adaptive adversary with 50% of the resource; 3) it achieves vertical scaling of throughput and confirmation latency to the physical limits of the participating nodes; 4) it achieves horizontal scaling with the number of nodes, enabling full decentralization. Trifecta comes in two versions: Proof-of-Work and Proof-of-Stake. A full-stack implementation in Rustachieves a throughput o $250,000$ transactions per second and a confirmation latency of the order of 20 to 30 seconds on a network of 100 EC2 nodes. This whitepaper explains the design and prototype of the system. Trifecta is built on top of Prism[4, 19].

1 Introduction

1.1 Democratizing trust at scale

Fundamental to the development of human society is cooperation, which is underpinned by trust. Modern platforms such as Uber, Airbnb, Visa, iTunes and Amazon marketplace act as digital trust intermediaries, providing trusted services at scale by large-scale data aggregation and processing. Since trust is such a basic primitive, it is not surprising that such platforms dominate modern life, as well as today's economy.

An unwanted feature of these platforms is the full centralization of trust into the intermediary, leading to several frictions. A natural direction of evolution of these platforms is the democratization of power and agency into the hands of the users of these platforms. Attempts to provide democratized services at scale have had a long history in the form of peer-to-peer services. Peer-to-peer data sharing networks such as BitTorrent dominated the internet traffic at some points in its development, but due to lack of formalized incentives and trusted outcomes, they lost out to centralized services.

Bitcoin heralded a new era of democratized trust, where a purely decentralized protocol guaranteed a secure ledger as well as formalized incentives by rewarding agents with trade-able tokens. This has created much excitement in the potential of creating decentralized trust services. However, the main pain point with Bitcoin, as well as later developments such as Ethereum, is the inability to scale performance efficiency.

We can visualize these different paradigms on a triangle (see Figure 1), where the three vertices are Democratization, Trust and Scale. Thus the existing paradigms operate on the sides of this triangle (for example Bitcoin achieves democratization and trust but not scalability and is therefore placed on the edge between those two attributes). There is no single paradigm achieving all three

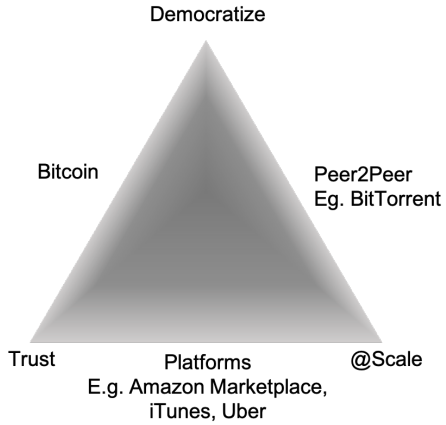


Figure 1: Existing Paradigms

properties. A protocol that can achieve all three properties delivers the best-of-all-worlds, thus democratizing trust at scale. Such a protocol enables trusted services to operate at scale without requiring centralized intermediaries.

1.2 The Blockchain Trilemma

Given the importance of achieving these properties in order to fulfil the promise of a trust revolution, designing a protocol that achieves all three properties has been the subject of recent interest. Several unsuccessful attempts to get all three properties has given rise to a folk theorem in blockchain, called the Blockchain Trilemma, first stated by .

The *Blockchain Trilemma* states that any blockchain can attain at most two of the following three properties.

1. Decentralization
2. Security
3. Scalability

In Figure 2, it is shown how different blockchain protocols fare in terms of the three properties. Bitcoin and Ethereum achieve full decentralization and security while being unable to scale. Attempts to scale blockchain such as EOS and Tron have led to centralization of the authority and trust through a small group of entities. Finally attempts to get decentralization and scalability in a blockchain system such as the GHOST protocol have proven to be insecure (for example, due to the balancing attack). The inability of any of these protocols to simultaneously achieve the three requisite properties is captured in the Blockchain trilemma.

1.3 Requirements

We now specify what it entails to achieve each of the three properties in greater detail. Firstly, we specify that a blockchain protocol maintains a ledger, which is a sequence of transactions or

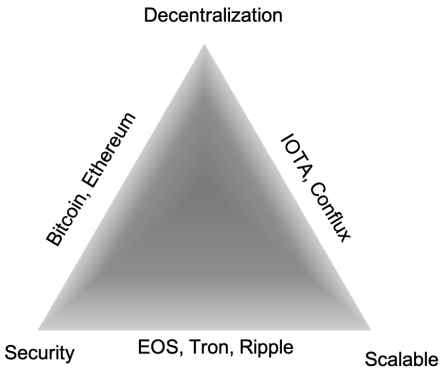


Figure 2: Blockchain Trilemma

instructions, which is held across a group of nodes. Authorized clients can issue transactions to be added to the ledger, and the protocol allows some nodes to propose and finalize changes to the ledger.

1. Decentralization

A blockchain protocol is said to be decentralized, if the power to propose or accept changes to the ledger is distributed to all nodes in the network. Such distribution of power can be in proportion to some resource that can be acquired by the nodes. One protocol is said to be more decentralized than another if it achieves equitable distribution over shorter time scale than another.

For example, in Bitcoin , power is distributed in proportion to the computational power held by the nodes. In a proof-of-stake system, power will be distributed proportional to the amount of stake (tokens) held by a node, which can be acquired by a node through a free market as required. Note that EOS is not decentralized since it concentrates power into a group of pre-approved nodes. A protocol such as Thunderella is less decentralized than Bitcoin since in Thunderella, the power to propose changes is confined to the leader unless the leader is voted out. A protocol is maximally decentralized if every edit (either proposal or acceptance) is done by a random node, whose identity is unpredictable before such an edit is issued. In this sense, Bitcoin is maximally decentralized.

2. Security

A protocol is said to be secure against adversaries if the consistency and liveness of the ledger can be guaranteed even if nodes holding together a certain fraction of resources collude maliciously. The resource would be compute power in a Proof-of-Work system and it would be stake in a Proof-of-Stake system. The maximum such fraction for which the protocol is secure is called the security threshold, β. We note that we will require the protocol to be secure against adversaries that adapt their controlled nodes based on the public state of the protocol (this provides a model for bribing adversaries that can decide the nodes-to-bribe based on public state).

3. Scalability

A protocol is said to be scalable if it scales performance efficiently. There are various performance metrics of interest that the protocol needs to optimize. We divide scalability into two aspects: (a) Vertical scaling: As the resources per node scales (such as computational power, storage, communication bandwidth and latency), how well does the performance of the blockchain scale. (b) Horizontal scaling: As the number of nodes scales, how does the performance of the blockchain scale.

Vertical scaling comprises various attributes, which we detail here.

1. Bandwidth efficiency: The bandwidth efficiency of a protocol is the ratio of the transaction rate supported by the protocol as a fraction of the available the network bandwidth. The ideal bandwidth efficiency is 1.

2. Latency: A protocol is said to be latency optimal if it can confirm transactions at a latency close to the latency of broadcast in the network.

Horizontal scaling refers to the ability of the protocol to scale the throughput of the blockchain linearly as the number of nodes increases (we will assume that the nodes are similar for simplicity). As the ledger size increases, the computation burden per node, storage requirement per node as well as communication load per node may increase.

By horizontal scaling, we mean that the throughput scales linearly in the number of nodes while the resources per node (computation, storage and communication) remain constant.

To summarize, an ideal blockchain protocol will have the following properties.

1. Maximal decentralization

2. Secure against adaptive adversaries

3. Scales both vertically and horizontally.

1.4 Trifecta solves the TriLemma

In this whitepaper we present Trifecta, a solution to the Trilemma. Trifecta has the following properties:

1. It is fully permissionless and decentralized.

2. It achieves full security, guaranteeing liveness and consistency of the ledger against a fully adaptive adversary with 50% of the resources.

3. It achieves vertical scaling of throughput and confirmation latency to the physical limits of the participating nodes.

4. It achieves horizontal scaling with the number of nodes, while the costs of computation, storage and communication per node remain constant.

5. It has a Proof-of-Work as well as a Proof-of-Stake version.

The solution is based on several key ideas:

- **block graph factorization**: By sortitioning blocks into multiple types of blocks of different functionality, security and scalability are decoupled, enabling Trifecta to achieve both simultaneously without a tradeoff.

- **coded Merkle tree**: The block graph factorization architecture is further exploited to achieve horizontal scaling by sharding without security compromise. The sharding solution achieves fully linear scaling of computation, storage and communication. The key problem of data availability is solved by coding of data blocks and the idea of *coded Merkle tree*.

- **work virtualization**: Proof-of-work systems can be made incentive compatible because the conservation of work incentivizes miners to mine on the longest chain rather on shorter chains. Proof-of-stake systems suffer from the *Nothing-at-stake* problem and miners have an incentive to mine on all chains to maximize the probability of proposing blocks. We introduce the idea of *work virtualization* which uses mining fees to simulate work conservation in a Proof-of-Stake system.

Trifecta is built on top of the Proof-of-Work permissionless protocol Prism[4]. Prism achieves vertical scaling of throughput and latency while having full security against an adversary with 50% hashing power. An open source full-stack implementation of Trifecta in Rustis reported in [19]. Trifecta is obtained by combining Prism with a sharding solution which in addition achieves horizontal scaling. We have also designed a Proof-of-Stake version of Trifecta.

We have implemented both versions in Rust. Figures 3 and 4 show the performance of the Proof-of-Stake version. A total throughput of around 250K transactions per second is achieved in a 6 shard system. The Proof-of-Work version has similar performance.

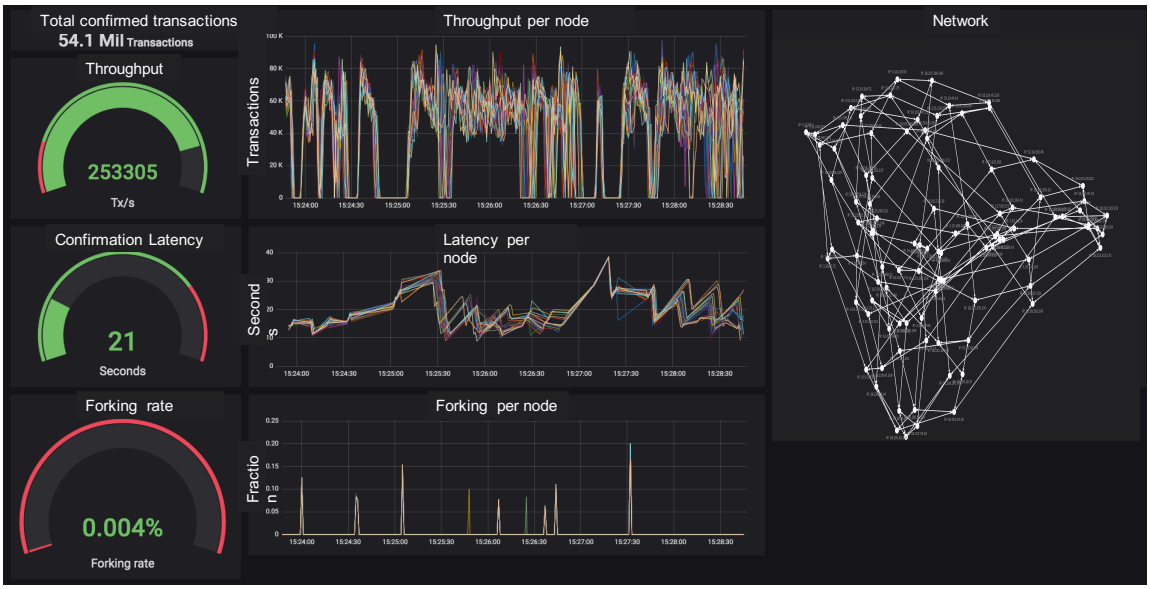


Figure 3: Total throughput, latency and forking rate.

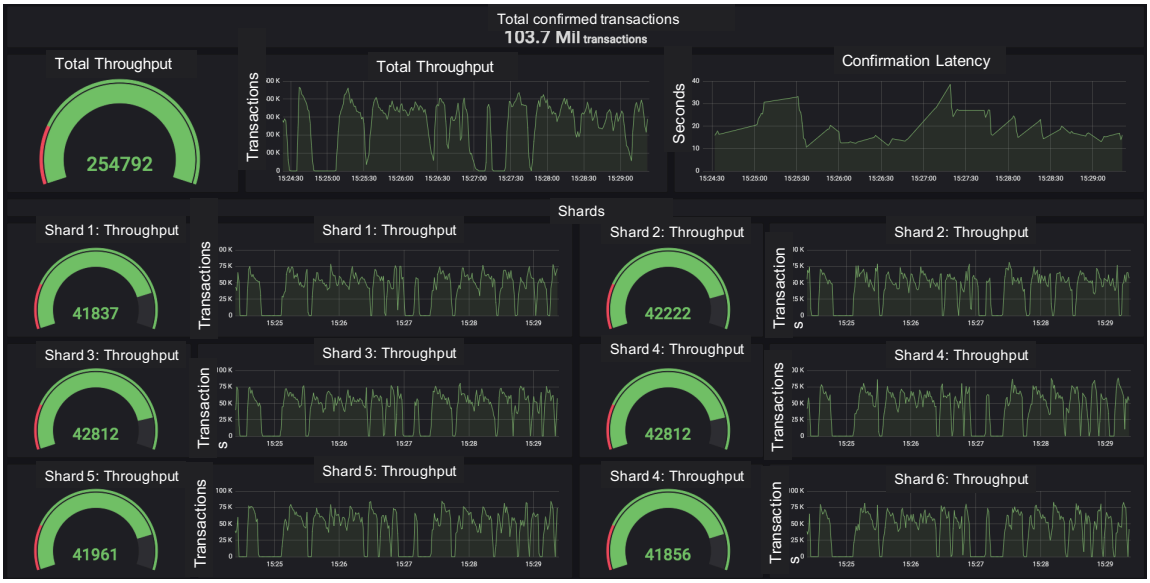


Figure 4: Throughputs of six shards.

In Section 2, we will discuss how vertical scaling is achieved in Prism. Section 3 discusses how Trifecta is designed by building a sharding solution on top of Prism. Section 4 discusses the implementation of Trifecta.

2 Prism: Vertical scaling via block graph factorization

In this section we will discuss the key idea of *block graph factorization* behind the design of Trifecta. We will start with Nakamoto's longest chain protocol and explore the limitations in scaling its throughput and latency. We will show how these limitations can be circumvented by removing the coupling between scalability and security in the longest chain protocol. This leads to the protocol Prism[4, 19]). It achieves full security and vertical scalability (throughput and latency at the physical limits of the nodes). We will sktech the basic ideas here; for full details of the protocol and security proofs of the protocol, please refer to [4]. In the next section, we show how the block graph factorization of Prism can be naturally combined with sharding to obtain Trifecta, which attains horizontal scaling as well.

2.1 The Longest Chain Protocol

The most basic blockchain consensus protocol is Nakamoto's longest chain protocol, used in many systems including Bitcoin and Ethereum. The basic object is a *block*, consisting of *transactions* and a reference link to another block. As transactions arrive into the system, a set of nodes, called *miners*, construct blocks and broadcast them to other nodes. The goal of the protocol is for all nodes to reach consensus on an ordered log of blocks (and the transactions therein), referred to as the *ledger*.

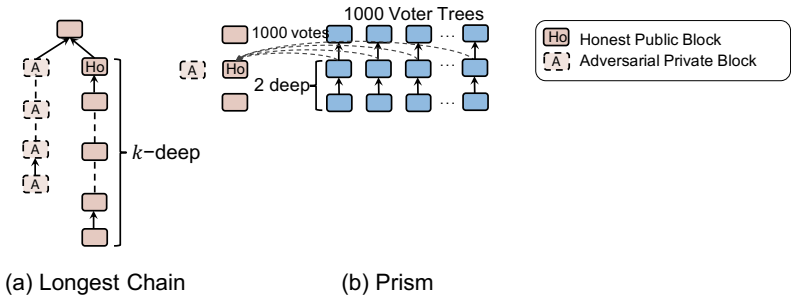


Figure 5: Depth of confirmation: longest chain vs. Prism . (a) The longest chain protocol requires a block Ho to be many blocks deep for reliable confirmation, so that an adversary mining in private cannot create a longer chain to reverse block Ho. (b) Prism allows each voter block to be very shallow but relies on many voter chains to increase the reliability.

Starting with the *genesis* block as the root, each new block mined by a miner is added to create an evolving *blocktree*. In the longest chain protocol, honest miners append each block to the leaf block of the longest chain in the current blocktree, and the transactions in that block are added to the transaction ledger maintained by the blocks in the longest chain. A miner earns the right to append a block after solving a cryptographic puzzle, which requires finding a solution to a hash inequality. The miner includes the solution in the block as a *proof of work* (PoW), which other nodes can verify. The time to solve the puzzle is random and exponentially distributed, with a mining rate f that can be tuned by adjusting the difficulty of the puzzle. How fast an individual miner can solve the puzzle and mine the block is proportional to its hashing power, i.e. how fast it can compute hashes.

A block is confirmed to be in the ledger when it is k-deep in the ledger, i.e. the block is on the longest chain and a chain of $k-1$ blocks have been appended to it. It is proven that as long as the adversary has less than 50% hashing power, the ledger has consistency and liveness properties [8]: blocks that are deep enough in the longest chain will remain in the longest chain with high probability, and honest miners will be able to enter a non-zero fraction of blocks into the ledger.

2.1.1 Latency Limitation

A critical attack on the longest chain protocol is the *private double-spend* attack [12], as shown in Figure 5(a). Here, an adversary is trying to revert a block Ho after it is confirmed, by mining a chain in private and broadcasting it when it is longer than the public chain. If the hashing power of the adversary is greater than that of aggregate of the honest nodes, this attack can be easily executed no matter what k is, since the adversary can mine blocks faster on the average than the honest nodes and will eventually overtake the public chain. On the other hand, when the adversary has less than half the power, the probability of success of this attack can be made exponentially small by choosing the confirmation depth k to be large [12]. The price to pay for choosing k large is increased latency in confirmation. For example, to achieve a reversal probability of 0.001, a depth of 24 blocks is needed if the adversary has $\beta = 30\%$ of the total hashing power [12].

2.1.2 Throughput Limitation

If B is the block size in number of transactions, then the throughput of the longest chain protocol is at most fB transactions per second (tps). However, the mining rate f and the block size B are constrained by the security requirement. Increasing the mining rate increases the amount of *forking* of the blockchain due to multiple blocks being mined on the same leaf block by multiple miners within the network delay Δ. Forking reduces throughput since it reduces the growth rate of the longest chain; recall that only blocks on the longest chain contribute to the ledger. More importantly, forking hurts the security of the protocol because the adversary requires less compute power to overtake the longest chain. In fact, the adversarial power that can be tolerated by the longest chain protocol goes from 50% to 0% as the mining rate f increases [8]. Similarly, increasing the block size B also increases the amount of forking since the network delay Δ increases with the block size [7].

A back-of-the-envelope calculation of the impact of the forking can be done based on a simple model of the network delay:

$$\Delta = \frac{hB}{C} + D,$$

where h is the average number of hops for a block to travel, C is the communication bandwidth per link in transactions per second, and D is the end-to-end propagation delay. This model is consistent with the linear relation between the network delay and the block size as measured empirically by [7]. Hence, the utilization, i.e. the throughput as a fraction of the communication bandwidth, is upper bounded by

$$\frac{fB}{C} < \frac{f\Delta}{h},$$

where $f\Delta$ is the average number of blocks "in flight" at any given time, and reflects the amount of forking in the block tree. In the longest chain protocol, to be secure against an adversary with $\beta < 50\%$ of hash power, this parameter should satisfy [8]

$$f\Delta < \frac{1-2\beta}{\beta}.$$

For example, to achieve security against an adversary with $\beta = 45\%$ of the total hashing power, one needs $f\Delta \approx 0.2$. With $h = 5$, this translates to a utilization of at most 4%. The above bound holds regardless of block size; the utilization of the longest chain protocol cannot exceed 4% for $\beta = 45\%$ and $h = 5$. Therefore, to not compromise on security, $f\Delta$ must be kept much smaller than 1. Hence, the security requirement (as well as the number of hops) limits the bandwidth utilization.

2.2 Prism: Factorizing the blockchain

The selection of a main chain in a blockchain protocol can be viewed as electing a leader block among all the blocks at each level of the blocktree. In this light, the blocks in the longest chain protocol can be viewed as serving three distinct roles: they stand for election to be leaders; they add transactions to the main chain; they vote for ancestor blocks through parent link relationships. The latency and throughput limitations of the longest chain protocol are due to the *coupling* of the roles carried by the blocks. Prism removes these limitations by factorizing the blocks into three types of blocks: proposer blocks, transaction blocks and voter blocks. (Figure 6). Each block mined by a miner is randomly sortitioned into one of the three types of blocks, and if it is a voter block, it will

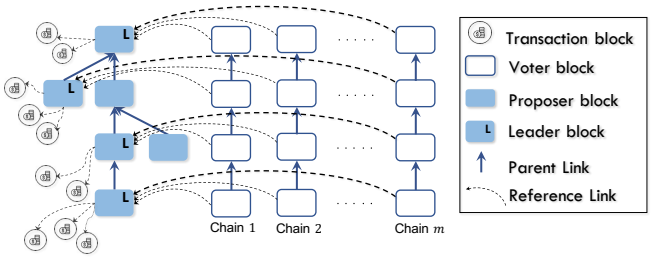


Figure 6: Prism : Factorizing the blocks into three types of blocks: proposer blocks, transaction blocks and voter blocks.

be further sortitioned into one of the voter trees. (Mining is described in detail in the pseudocode in Appendix A.)

The proposer blocktree anchors the Prism blockchain. Each proposer block contains a list of reference links to transaction blocks, which contains transactions, as well as a single reference to a parent proposer block. Honest nodes mine proposer blocks on the longest chain in the proposer tree, but the longest chain does not determine the final confirmed sequence of proposer blocks, known as the *leader sequence*. We define the *level* of a proposer block as its distance from the genesis proposer block, and the *height* of the proposer tree as the maximum level that contains any proposer blocks. The leader sequence of proposer blocks contains one block at every level up to the height of the proposer tree, and is determined by the *voter chains*.

There are m voter chains, where $m \gg 1$ is a fixed parameter chosen by the system designer. The ith voter chain is comprised of voter blocks that are mined on the longest chain of the ith voter trees. A voter block votes for a proposer block by containing a reference link to that proposer block, with the requirements that: 1) a vote is valid only if the voter block is in the longest chain of its voter tree; 2) each voter chain votes for one and only one proposer block at each level. The leader block at each level is the one which has the highest number of votes among all the proposer blocks at the same level (tie broken by hash of the proposer blocks.) The elected leader blocks then provide a unique ordering of the transaction blocks to form the final ledger.

A pseudocode of the protocol can be found in Appendix A.

2.2.1 Security and Latency

The votes from the voter trees secure each leader proposer block, because changing an elected leader requires reversing enough votes to give them to a different proposer block in that level. Each vote is in turn secured by the longest chain protocol in its voter tree. If the adversary has less than 50% hash power, and the mining rate in each of the voter trees is kept small to minimize forking, then the consistency and liveness of each voter tree guarantee the consistency and liveness of the ledger maintained by the leader proposer blocks. However, this would appear to require a long latency to wait for each voter block to get sufficiently deep in its chain. What is interesting is that when there are many voter chains, the same guarantee can be achieved without requiring each and every vote to have a very low reversal probability, thus drastically improving over the latency of the longest chain protocol.

To get some intuition, consider the natural analog of the private double-spend attack on the longest chain protocol in Prism . Figure 5(b) shows the scenario. An honest proposer block Ho at a particular level has collected votes from the voter chains. Over time, each of these votes will become deeper in its voter chain. An attack by the adversary is to mine a private proposer block

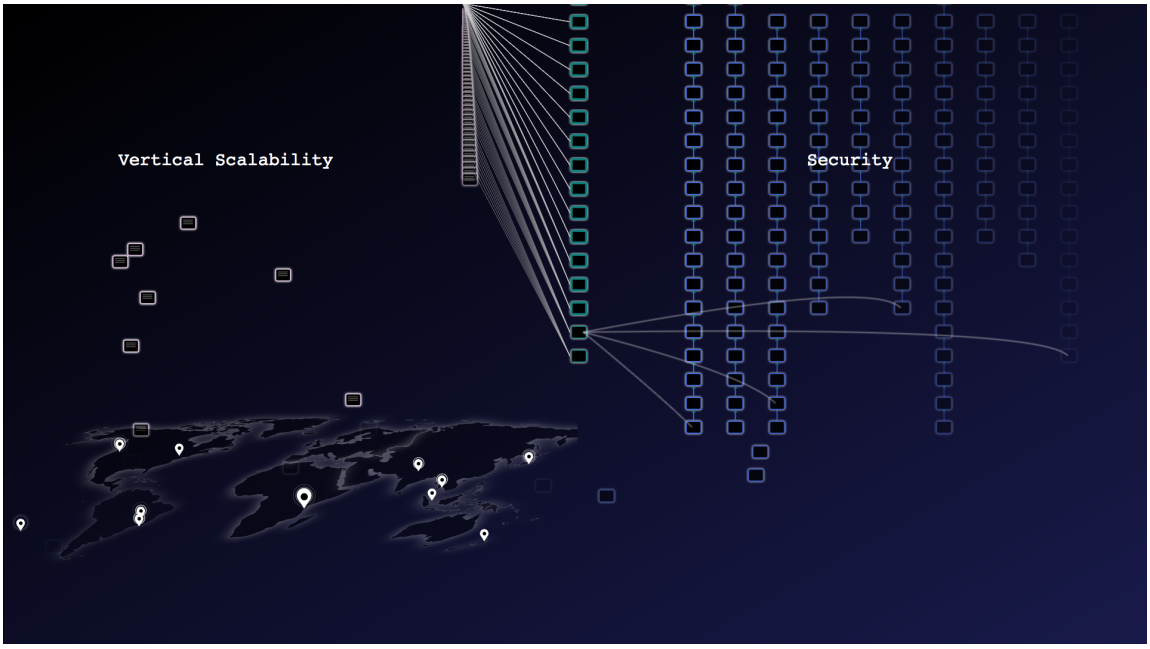


Figure 7: Prism in action.

A at the same level, and on each of the voter trees, fork off and mine a private alternate chain and send its vote to block A. After leader block Ho is confirmed, the adversary continues to mine on each of the alternate voter chains to attempt to overtake the public longest chain and shift the vote from Ho to A. If the adversary can thereby get more votes on A than on Ho, then its attack is successful. The question is how deep do we have to wait for each vote to be in its voter chain in order to confirm the proposer block Ho?

Nakamoto's calculations will help us answer this question. As an example, at tolerable adversary power $\beta = 30\%$, the reversal probability in a single chain is 0.45 when a block is 2-deep [12]. With $m = 1000$ voter chains and each vote being 2-deep, the expected number of chains that can be reversed by the adversary is 450. The probability that the adversary can get lucky and reverse more than half the votes, i.e. 500, is about 0.001. Hence to achieve a reversal probability, $\varepsilon = 0.001$, we only need to wait for the votes to be 2-deep, as opposed to the 24 block depth needed in the longest chain protocol (Section 2.1.1). This reduction in latency comes without sacrificing security: each voter chain can operate at a slow enough mining rate to tolerate β adversarial hash power. Furthermore, increasing the number of voter chains can further improve the confirmation reliability without sacrificing latency; for example, doubling the number of voter chains from 1000 to 2000 can reduce the reversal probability from 0.001 to 10^{-6}.

We have discussed one specific attack, focusing on the case when there is a single public proposer block on a given level. Another possible attack is when there are two or more such proposer blocks and the adversary tries to balance the votes between them to delay confirmation. It turns out that the attack space is quite huge and these are formally analyzed in [4] to obtain the following guarantee on the confirmation latency, regardless of the attack:

Theorem 2.1 (Security abnd Latency, Thm. 4.8 [4]). *For an adversary with $\beta < 50\%$ of hash power, network propagation delay D,* Prism *with m chains confirms honest*[1] *transactions at reversal probability ε guarantee with latency upper bounded by*

$$Dc_1(\beta) + \frac{Dc_2(\beta)}{m} \log \frac{1}{\varepsilon} \quad \textit{seconds}, \tag{1}$$

where $c_1(\beta)$ and $c_2(\beta)$ are β dependent constants.

For large number of voter chains m, the first term dominates the above equation and therefore Prism achieves near optimal latency, i.e. proportional to the propagation delay D and independent of the reversal probability. Note that (1) is a worst-case latency bound that holds for *all* attacks.

2.3 Throughput

To keep Prism secure, the mining rate and the size of the voter blocks have to be chosen such that each voter chain has little forking. The mining rate and the size of the proposer blocks have to be also chosen such that there is very little forking in the proposer tree. Otherwise, the adversary can propose a block at each level, breaking the liveness of the system. Hence, the throughput of Prism would be as low as the longest chain protocol if transactions were carried by the proposer blocks directly.

To decouple security from throughput, transactions are instead carried by separate transaction blocks. Each proposer block when it is mined refers to the transaction blocks that have not been referred to by previous proposer blocks. This design allows throughput to be increased by increasing the mining rate of the transaction blocks, without affecting the security of the system. The throughput is only limited by the computing or communication bandwidth limit C of each node, thus potentially achieving 100% utilization. In contrast, as we discussed in Section 2.1.2, the throughput of the longest chain protocol is security-limited, resulting in low network utilization. [4] formally proves that Prism achieves near optimal throughput:

Theorem 2.2 (Throughput, Thm. 4.4[4]). *For an adversary with $\beta < 50\%$ fraction of hash power and network capacity C,* Prism *can achieve $(1-\beta)C$ throughput and maintain liveness in the ledger.*

Note that the reference links from a proposer block to its transaction blocks are sealed in the proof-of-work of the proposer block. Hence, once a proposer block is mined, there is no way for an attacker to change the reference links to the transaction blocks. In contrast, the transactions that a miner in Bitcoin -NG adds onto the ledger after its block enters the main chain are *not* sealed by the proof-of-work. Hence, Bitcoin -NG is susceptible to censorship attacks.

2.4 From Proof-of-Work to Proof-of-Stake

The security of Prism is maintained by the voter chains. In turn, the security of each of the voter chains is maintained by the Proof-of-Work longest chain protocol. To obtain a Proof-of-Stake version of Prism , it is natural to mimic Nakamoto longest chain protocol in a Proof-of-Stake setting. This is achieved in [18], which designed and proved security guarantees of such a protocol.

[1] Honest transactions are ones which have no conflicting double-spent transactions broadcast in public.

3 Trifecta: solving trilemma via sharding

Conjecture 3.1 (Blockchain Trilemma)**.** *blockchain system can have only two of the following three properties*

1. ***Decentralization:*** *Each participant in the system has access to only $O(c)$ resources i.e, a regular laptop*
2. ***Scalability:*** *The system can process $O(n) > O(c)$ transactions.*
3. ***Security:*** *Defend against an adversary with $O(n)$ resources.*

Using graph factorization in Prism, we achieved vertical scaling to reach the physical limits of the underlying peer-to-peer network. On top of Prism, we develop the protocol Trifecta which to achieve horizontal scalingb as well, hence solving the Trilemma. The overall throughput of Trifecta scales with the number of participating nodes, each of which has a fixed amount of computation, storage, and communication resources.

Trifecta achieves horizontal scaling via *sharding*, which is a technique to increase the throughput by running multiple blockchains (called shards) in parallel, where each shard keeps track of a disjoint set of accounts, and each node is allocated to a particular shard. If the number of nodes processing a given shard is fixed to be a constant, then the number of shards can be increased linearly as more nodes join the system. This automatically scales the throughput of the system while ensuring that computational load, communication burden and storage per node remain constant.

3.1 Trifecta sharding protocol

Trifecta utilizes a simple mechanism to allocate nodes to shards, based on *self-allocation*, i.e., nodes allocate themselves to shards rather than being allocated by some complicated algorithms proposed in prior sharding solutions [2, 11, 9]. If nodes self-allocate to shards, then it seems that it is impossible to preserve high security - since adversarial nodes could very well congregate on a given shard. We seek a sharding design where security is not at all dependent on the nodes allocated to the particular shard but based on the size of the entire network.

To achieve this security, every node would have to validate every transaction in every shard making it infeasible to scale. A key idea behind the Trifecta sharding scheme is that validation is decoupled from consensus, i.e., all nodes agree on the sequence of transactions for each shard but not all transactions need to be valid. This is also the key idea behind the design of sharded distributed systems [5, 6] secure against crash faults. To get the correct shard state, any shard node can download the shard ledger and processes transactions in the shard (discarding any invalid transactions) to get the finalized state (we call this process ledger sanitization). Now that validation is not inherent in consensus, Trifecta can get computational efficiency (since only shard nodes execute transitions in the ledger). In order to make sure that every node does not store every block, we begin with Prism and allow transaction blocks to be colored, i.e., to belong to different shards.

We will now formally describe the sharding protocol. To begin with, we will assume (a) that all blocks are communicated to everyone and (b) that there are no inter-shard transactions (e.g., K different applications each running on a separate shard). We will relax these two assumptions in upcoming sections.

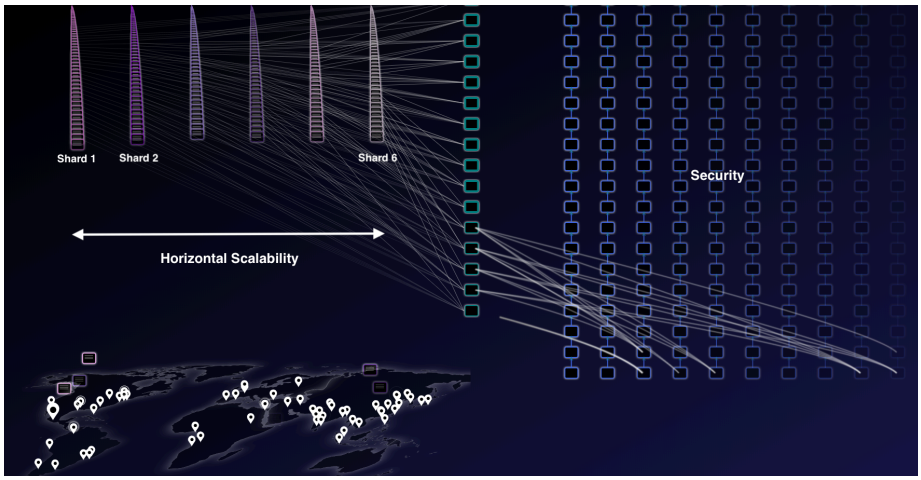


Figure 8: Trifecta. The proposer blocks (green) and the chains of the voter blocks (blue) are maintained by every node. Transaction blocks (red) are partitioned into $K = 6$ shards, each of which is represented as a column on the left. Each node of shard i only stores and validates the transaction blocks in its own shard. A proposer block refers to transaction blocks from all shards. All the referred transaction blocks are confirmed once the proposer block receives sufficiently many votes from the voter chains. When ordering the confirmed transaction blocks, each node only uses the transaction blocks in its own shard to construct the shard ledger.

Suppose there are N decentralized nodes in the network. Trifecta sharding protocol will set up $K = \Theta(N)$ shards, where K is chosen to scale linearly with the network size N. Each shard maintains a ledger of transactions that are independent of the other shards. Each node, based on its own interest, individually selects one of the K shards to join. Once joining a shard, a node is responsible for validating and storing transactions belonging to that shard.

Trifecta sharding protocol is a natural extension of the graph factorization concept described in Section 2. Each node, no matter which shard it belongs to, maintains the *same* tree of proposal blocks, and the *same* m voter chains. This is a fixed amount of overhead to maintain security. However, we scale the total number of transaction blocks, which are partitioned into a large number of distinct shards (Figure 8). Each transaction block contains an additional field that indicates its shard ID. Each node is trying to mine proposer and voter blocks, but is only interested in mining transaction blocks within its own shard.

Once a transaction block is generated, it is circulated among all the nodes. The nodes who belong to the shard ID indicated in the transaction block will store that transaction block. Nodes from other shards will download the block but only temporarily store it to ensure data availability (at a honest shard node). Consequently, each node only permanently stores the transaction blocks in its own shard. When creating a proposer block, a miner will include the references to all transaction blocks (in its shard or outside its shard), and broadcast the generated proposer block to the entire network. The voter blocks are created the same as in Prism, whose contents are votes on proposer blocks.

Note that proposer blocks which point to transaction blocks for which data is unavailable will be considered invalid. After a proposer block collects enough votes and gets confirmed, all transaction

blocks for all shards referred by this proposer block are also confirmed. Each node only uses the subset of these transaction blocks that lie in its own shard to construct a valid ledger for their shard.

The voting and confirmation rules exactly follow the Prism protocol. Therefore, Trifecta inherits the security guarantee of being robust to 50% adversarial mining power. Note that compared with other well-known sharding solutions, Trifecta sharding does not require honest majority within each shard. This is because reversal of a confirmed shard transaction requires reversal of the confirmed proposal block, and this requires a majority of voting power to accomplish. We note that, **even when there are no honest miners in a shard**, the shard remains consistent. For liveness, Trifecta only requires one honest node in a shard in order to continue generating honest transaction blocks. Therefore, our protocol remains secure even when the 50% adversary are adaptive. This is a consequence of the decoupling between security and scalability in the graph factorization architecture.

Within each shard, all the nodes belonging to this shard generate transaction blocks to saturate the physical bandwidth limits (vertical scaling). Across different shards, the transaction blocks are independently generated, so that the overall system throughput equals to the summation of the throughput of all shards. Moreover, since each node only verifies and stores the transactions blocks in its own shard, it is obvious that the loads of computation and storage do not increase with the number of shards. To summarize, the advantages of Trifecta include the following:

1. **Decentralization:** The system can work in both Proof-of-Work as well as Proof-of-Stake networks. In Proof-of-Work system, our sharding solution does not require any identity management (which is required for solutions relying on node-to-shard rotation).

2. **Security:** The system is fully secure against adversaries that can adaptively (by bribing or otherwise) corrupt a sub-minority of computational power. In particular, our scaling solution provides full safety even when the entire shard is corrupt, as long as a majority of the total compute power is honest. The shard is live (will produce new honest transactions) as long as there is one honest node per shard.

3. **Scalability:** The system provides linear scaling of throughput with respect to the number of shards. Since each shard only requires one honest node, the number of shards can be proportional to the number of honest nodes. This provides high scalability compared with protocols that require each shard to be large enough to have majority honest miners. The computational and storage resources per node do not grow as the number of shards increases, thus the protocol is computation and storage efficient.

Beacon shard: Finally, in addition to the K types of transaction blocks each corresponding to a particular shard, we also create a special type of shard-independent transaction block, which is called a beacon transaction block. We can think of this block as belonging to a special shard, called the beacon shard, that every node in the network belongs to. Thus there is a beacon ledger, which is maintained by all the nodes. The beacon ledger is used for storing state required for all the nodes - for example, rewards to incentivize proposer and voter blocks are maintained in the beacon shard. We note that it is easy for a node to maintain any arbitrary subset of shards, in which it is interested (in addition to the beacon shard). However, when producing a transaction block, the node will choose a particular shard ID and select transactions for that particular shard.

Next, we briefly discuss the problems of the node-to-shard allocation strategies in the existing sharding protocols and contrast with our approach. Following that, we explain the "data avail-

ability" problem, and describe how our Trifecta sharding protocol can be extended to deal with it to maintain constant communication load at each node. Then we extend the protocol to allow inter-shard transactions.

Prior sharding schemes

Existing blockchain sharding solutions (see, e.g., [2, 11, 9]) maintain the system security by requiring each shard to be secure individually. However, even for a network of nodes whose majority are honest, it is relatively easy for an adversary to attack a single shard. Therefore, almost all prior sharding solutions focus on patching this security hole by developing a good node-to-shard allocation mechanism, where nodes are allocated to shards using a (cryptographically) randomized algorithm and they rotate these nodes across shards in every 1-2 days (or less). The idea is that if the original network has a minority of adversarial nodes, each shard also likely has a minority of adversarial nodes, thus making each shard secure. There are three key difficulties in this approach.

1. **Decentralization**: The node-to-shard allocation requires maintenance of node identities in the system - this is antithetic to the identity-free nature of PoW blockchains. This gives a strong degree of predictability to the shard blocks - they can never be generated by nodes not allocated to the shard. Furthermore, in both PoW and PoS systems, a secure node-to-shard allocation may require new and costly secure multiparty computation or equivalent protocols which has a large overhead.

2. **Security**: The system is not secure against adaptive adversaries or to bribes. An adversary can immediately corrupt/bribe the nodes after they have been allocated to a shard, hence the obtaining majority within particular shard(s). The existing proposals try mitigating this by rotating the nodes across shards but that imposes a huge engineering and incentive challenge where a node has to fully download the state of a shard after each allocation thus increasing the overhead tremendously and severely limiting the frequency of rotation.

3. **Scalability**: Even in the absence of adaptive adversaries, the size of each shard has to be large. This is because the node-to-shard allocation creates a random subset of nodes for each shard, and for this subset to be representative of the overall population requires high number of nodes. This places a lower bound on the number of nodes in the shard to acquire a certain adversarial protection. For example, in Eth 2.0 sharding, they require each shard to contain 100's of nodes so that each shard is representative. While such a system can achieve performance increasing with the number of shards, with N nodes there can be at most $N/100$ shards, thus highly limiting its value. For example, for a network of 800 nodes, Eth 2.0 sharding can only accommodate 8 shards.

3.2 Data availability problem and erasure codes

In the present version of the protocol, every transaction block is sent to everyone. This implies that the communication per node increases with the number of shards. A naive mechanism to reduce the communication requirement of this sharding scheme is to transmit the transaction block only to its own shard (on a shard-specific peer-to-peer network) but transmit the header to all the nodes.

However, this implementation can cause validity issues on the proposer blocks. An obvious attack is for the adversary to only send the header to every node in the network without sending the content to the shard node - thus no honest node has access to this transaction block. Now, if

a proposer block that includes a link to this transaction block gets confirmed, then the processing of the particular shard can completely stall. In the present version of the protocol, this possibility is precluded by requiring every node to download the block as well as storing it temporarily before considering it to be valid. In order to keep the communication requirement fixed as the number of shards increases, we require a communication efficient way to check for data availability.

As such motivated, we phrase the following *data availability* problem: **Data availability problem:** Given only the header of some transaction block $\mathbf{B}$ in shard j, how can a node in shard i verify that the data is available with at least one honest node in shard j in a *communication efficient* manner?

A key question is how to check the data is available at a *honest* node. We make the following minimal assumption: that each node in the network is connected to *at least one honest node in every shard*. We note that this assumption is much weaker than assuming that majority of each shard is honest.

Instead of downloading the entire block, a node could randomly sample a small portion of $\mathbf{B}$, from some node in shard j who claims to have the entire block $\mathbf{B}$. For the purpose of data availability, we can simply view $\mathbf{B}$ as a stream of b bytes. For some design parameter k, we can evenly and arbitrarily partition the block into k chunks $m_1, \cdots, m_k$, each of size $\frac{b}{k}$ bytes. We call each of these chunks a data symbol. When one of the k symbols is missing (or intentionally hidden by malicious block producers), a node in a different shard i will hit this symbol with probability $\frac{1}{k}$ when sampling uniformly at random. Consequently, it will take the node on average k trials before finding out the block is not available, and this reduces to the inefficient approach of downloading the entire block.

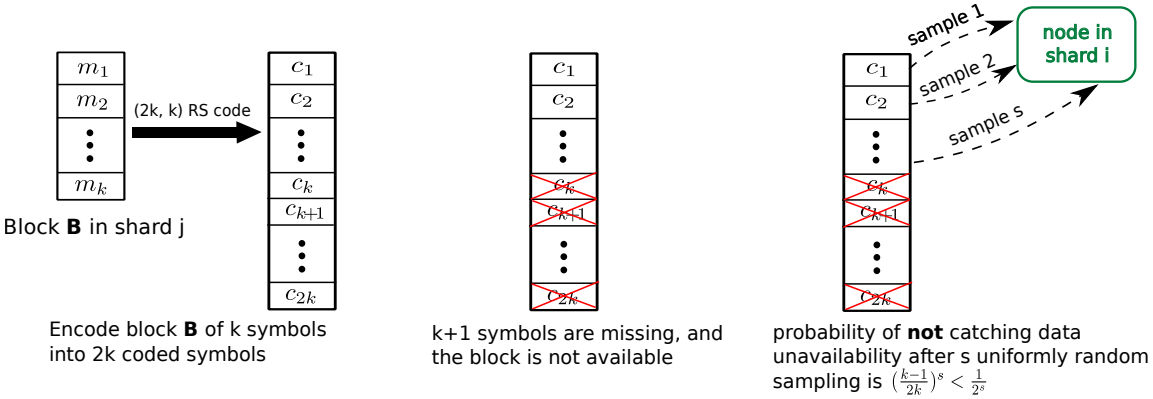


Figure 9: Random sampling to check data availability of a block $\mathbf{B}$, which is encoded by a $(2k, k)$ Reed-Solomon code.

Improving sampling efficiency with erasure codes

Similar data availability problem was encountered in the framework of light clients, where a light client who only stores the headers of blocks (say, e.g., an SPV client for Bitcoin [1]) tries to verify the availability of the entire block. It was proposed in [3] to improve the sampling efficiency of light clients by encoding a block using error-correcting codes (see, e.g., [10]). Here, we can also adopt

this technique to reduce the amount of data a node needs to download to verify the availability of a block in another shard.

To illustrate the coding idea, let us apply an (n, k) Reed-Solomon (RS) code [14] to the block $\mathbf{B}$ to create $n \geq k$ coded symbols $c_1, \cdots, c_n$ from the uncoded ones $m_1, \ldots, m_k$. Using this code, $\mathbf{B}$ can be recovered using *any* k out of n coded symbols. Therefore, the block is available unless $n - k + 1$ coded symbols are missing. When for instance $n = 2k$, as illustrated in Figure 9, after successfully sampling s coded symbols, a node in shard i gains confidence that with probability $1 - (\frac{k-1}{2k})^s$ some node in shard j has at least 50% of the coded symbols so it can completely recover $\mathbf{B}$. In terms of communication cost, the node in shard i now only needs to download on average 2 symbols before concluding that $\mathbf{B}$ is unavailable.

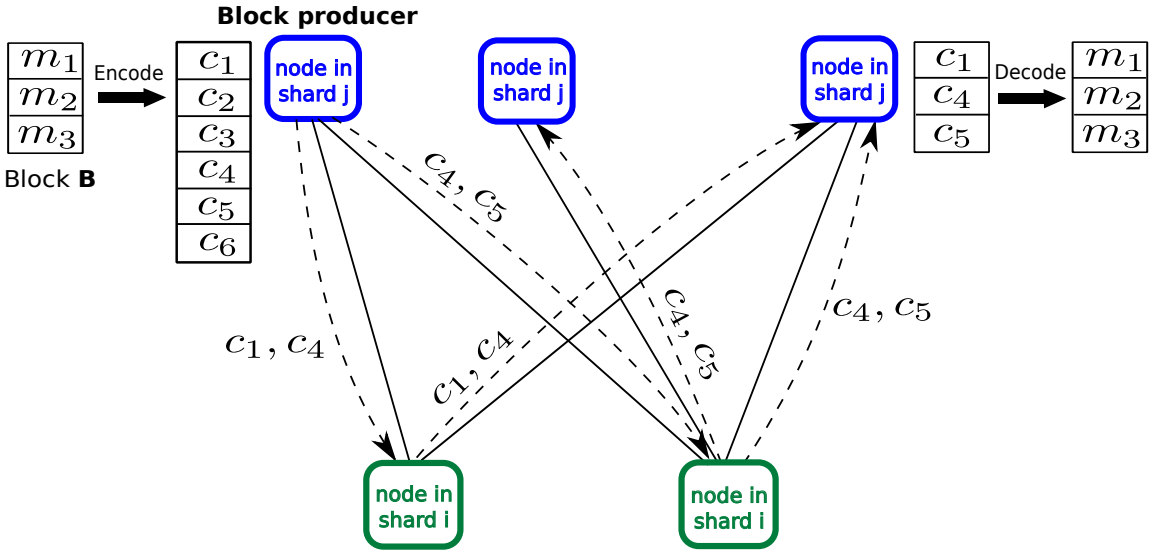


Figure 10: To verify the availability of a block $\mathbf{B}$ in shard j, a node in some other shard i randomly samples some coded symbols of $\mathbf{B}$ and forwards them to its neighbours in shard j. One of the honest nodes in shard j uses the coded symbols to decode $\mathbf{B}$, and verifies the correctness of coding.

While a node in shard i gains confidence via random sampling that some node in shard j will have sufficiently many symbols to decode the block, it cannot verify if these coded symbols are generated correctly, or in other words if decoding them will indeed yield the original $\mathbf{B}$. To defend such "incorrect-coding" fraud, the nodes outside the shard will forward their sampled symbols to their neighbouring nodes inside the shard, and rely on them to detect coding errors (Figure 10).

Coding-fraud proof

An honest node in shard j decodes the block $\mathbf{B}$ using coded symbols sampled by nodes in foreign shards, and verifies that the encoding was done correctly by the block producer. To do that for the above RS code, we can have the block producer create a Merkle tree from all coded symbols and store the commitment (Merkle root) in the block header. Having decoded a block, a node can re-construct the commitment, and compare with the one stored in the header to verify coding correctness. Once a coding error is detected, the node should construct a coding-fraud proof to

inform the other nodes in shard j, so they can safely ignore this transaction block when constructing the ledger. To create such proof, one node will need to include k coded symbols using which the original block can be decoded and verified. However, this yields a proof size that is as large as the entire block. To reduce the proof size, it was proposed in [3] to use 2-dimensional Reed-Solomon (2D-RS) codes, for which the k symbols in a block is arranged into a $\sqrt{k} \times \sqrt{k}$ square, and then each row/column is encoded into $\sqrt{n}$ coded symbols using a RS code. This reduces the proof size from k to $\sqrt{k}$ symbols, since now one only needs to show that one row (or column) is incorrectly encoded.

In the Trifecta sharding protocol, when a coding-fraud proof is generated, it is included in the next proposer block as a special transaction. Now, since each proposer block is downloaded by every node in the network, when the size of a transaction block scales with the number of shards K (hence number of nodes N), the communication cost at a single node to download the coding-fraud proofs generated according to [3] will scale as $O(\sqrt{N})$, and blow up as network expands. Hence, to make Trifecta communication efficient, we need a mechanism to verify data availability with 1) constant sampling size for foreign transaction blocks, and 2) constant coding-fraud proof size.

3.3 Communication efficiency: coded Merkle tree

An (n, k) linear code is completely characterized by a $(n-k) \times n$ parity check matrix $\mathbf{H}$, such that a transaction block is correctly encoded iff the coded symbols $c_1, \ldots, c_n$ satisfies $\mathbf{H}[c_1, \cdots, c_n]^\top = \mathbf{0}$. In other words, the coded block has to pass $n-k$ parity check equations, each of which is an inner product with a row of $\mathbf{H}$. Since an incorrectly encoded block fails at least one of these equations, we propose to use a *single* failed equation as the coding-fraud proof. To make this proof succinct (ideally constant size) and verifiable, our design needs to meet the following requirements:

1. The parity check matrix $\mathbf{H}$ needs to be sparse. More precisely, each row of $\mathbf{H}$ should contain a constant number of 1s, such that each equation involves constant number of symbols.

2. The decoder should be able to identify a particular unsatisfied equation, and prove it to the other nodes assuming they only have the header of the block.

We leverage regular LDPC codes for which each parity check equation involves computing XOR of a *constant* d coded symbols. When one of the equations is violated, the decoder generates a coding-fraud proof consisting of $d-1$ symbols and d membership proofs for all the symbols in the equation (we will discuss how to construct cryptographic commitment for the coded block shortly). Another node can simply verify the proof by 1) verifying that the $d-1$ symbols pass their membership proofs, and 2) decoding the missing symbol, and verifying that it fails its membership proof. To enable such proof with constant size of $d-1$, Trifecta sharding solution adopts a hash-aware peeling decoder, which decodes a new symbol from an equation with $d-1$ hash-verified symbols at each time. The decoding process stops once all the symbols are decoded, or if no parity equation exists with $d-1$ available symbols. In the latter case a decoding error is declared. With peeling decoding, a fraud proof can be readily generated once the newly decoded symbol from some equation does not match its hash commitment.

We use particular ensembles of LDPC codes that are known to perform well under the peeling decoder (see, e.g., [13]), i.e., LDPC codes with large "stopping distance" which are guaranteed to be decodable even if only a constant fraction of the coded symbols are available. As an example, an $(n, k) = (32000, 8000)$ regular LDPC code whose parity check equation involves $d = 8$ coded

symbols can be decoded by a peeling decoder from any $0.876n = 28032$ symbols, with probability of at least 99.97%. This also makes the probability that the block is not available, but not detected by a node after $s = 30$ samples as small as 0.00063%. Therefore, a node outside the shard takes a constant number of samples to check data availability.

To be able to verify each decoded symbol in every iteration of the peeling decoder, we require the hash commitments of *all* coded symbols available at the decoder. Storing all of them in the block header is prohibitively expensive since the number of hashes increases with block size n. To resolve this issue, instead of storing these hashes, we propose to recover them on-the-fly during the decoding process. Recovering the n hashes for the coded symbols naturally induces another data availability problem, which can be solved by applying erasure codes on the hash data. Motivated by this idea, we propose a novel cryptographic accumulator named coded Merkle tree (CMT), which iteratively encodes the data with an LDPC code, computes hash commitments of the coded symbols, and concatenates these hashes into the data for the next level, until the size of the hash commitments is small enough to be stored in the block header.

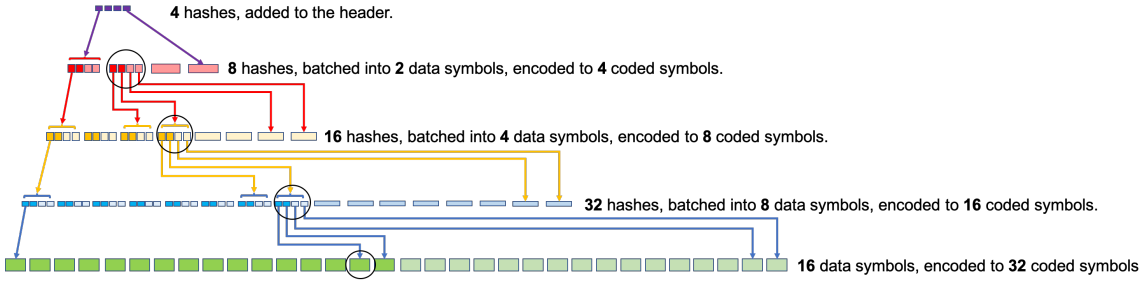


Figure 11: Coded Merkle tree construction for $k = 16$ data symbols with a rate $r = \frac{1}{2}$ code. The hashes of the $t = 4$ symbols in the last level are stored in the block header. The set of circled symbols form the membership proof of the circled symbol on the base level.

Coded Merkle tree. The first level of the tree consists of n coded symbols, created from k data symbols constituting the original block via a rate $r = k/n$ systematic LDPC code. For the second level, we first compute the hashes of all the symbols in the first level, then aggregate q hashes together to form a total of n/q new data symbols, for some design parameter q, and finally use another $(\frac{n}{qr}, \frac{n}{q})$ LDPC code to generate $\frac{n}{qr}$ coded symbols. We repeat this process to grow higher levels of the tree, until we reach a constant target number of t symbols at level $1 + \log_{qr} \frac{n}{t}$. Finally, we compute t hashes of the symbols in the last level, and store them in the block header.[2] As an example, we illustrate in Figure 11 the construction of a coded Merkle tree for a block of $k = 16$ data symbols, encoded with rate $r = \frac{1}{2}$. Every $q = 4$ hashes of the coded symbols are grouped into a new data symbol on the next level. After the construction, only $t = 4$ hashes are included in the block header. Similar to the Merkle proof of a regular Merkle tree, the membership proof of a base level symbol in CMT consists of all the sibling hashes between this symbol and the tree root. The only difference is that the number of sibling hashes per level is now $q - 1$ instead of 1. We note that a membership proof of a symbol automatically provides proofs for all the symbols on the higher levels in the proof. Hence, sampling a symbol on the base level (together with its membership proof) samples one membership-proved symbol on each level of CMT for free. Interested readers

[2] A regular Merkle tree is a special case of the coded Merkle with $r = 1$, $q = 2$, and $t = 1$.

are referred to [20], for further details on the selections of LDPC codes on each level, and the exact construction of CMT.

To decode a block, a node decodes the corresponding CMT starting from the top level, and all the way down to the base level. The decoded data at each level, which is the hash commitments of the symbols on the level below, is used to verify the symbols that are newly decoded from the peeling decoder. Utilizing CMT, a node can construct a coding-fraud proof at any level with a constant size of $d-1$ symbols. Compared with a conventional Merkle tree, the proposed CMT enables succinct coding-fraud proof while maintaining a small size for the tree commitment.

We note that using CMT, our Trifecta sharding protocol is truly efficient in storage and computation since 1) each block stores a constant number of CMT root hashes in the header, 2) the decoding complexity of a peeling decoder is linear in block size.

Constant communication load

As we mention before, using a peeling decoder for an LDPC code achieves high sampling efficiency, such that each node only needs to sample a constant number of symbols before detecting the unavailability of a block. Let us say a constant fraction α of the coded symbols need to be received to decode the block, and each node randomly samples a constant s number of symbols uniformly at random. Then on average we will need samples from $N = \frac{\alpha n \log(\alpha n)}{s} \approx O(n)$ nodes. Therefore, the number of symbols in each block n also scales with the network size N. Recall that running Trifecta for a single shard already achieves the maximum throughput subject to the underlying compute network, scaling up the size of a block means that the number of generated blocks per second in a single shard scales as $O(\frac{1}{N})$. Now, let us consider the communication load at a node that consists of two parts:

- Sampling cost: it takes s symbols from each of the transaction blocks in the $K-1=\Theta(N)$ foreign shards. This leads to a constant $O(\frac{s(K-1)}{N}) = O(1)$ communication rate.
- Coding-fraud proof cost: it downloads $d-1$ symbols for each coding-fraud proof from a shard, which incurs a total communication rate of $O(\frac{(d-1)K}{N}) = O(1)$.

This demonstrates that our Trifecta sharding protocol is communication-efficient.

3.4 Inter-shard transactions

We have described so far the Trifecta sharding protocol for the scenario where transactions only occur within each shard. Inter-shard transactions are enabled in Trifecta utilizing the technique of "state commitments", which we describe in the following.

State commitments

In order to run light nodes that can verify and process transactions without requiring to download the entire state of a shard, we provide an additional functionality called state commitments. The state $S_{s,\ell}$ of a shard chain s till a certain level ℓ of the proposer block tree is simply the overall state of the ledger, i.e., it contains details about all memory items after executing the instructions contained till the leader proposer block of that level. For example, in a UTXO (unspent transaction output) system, the state is simply the list of all unspent outputs after processing shard chain transactions until the leader on level ℓ.

The state commit contains, among other things, roots of the Merkle tree (or more generally any commitment structure like a Merkle Particia tree or RSA acculumator) constructed from the state $S_{s,\ell}$ of shard s till the leader proposer block at level ℓ. We will term $c_{s,\ell} := \text{Commit}(S_{s,\ell})$ as the state commit of shard s at level ℓ.

Since there are potentially a majority of malicious nodes in any given shard, additional mechanisms are required to validate the state commit - they cannot be assumed to be correct. Any node can post state commitments based on some Sybil resistant mechanism (like Proof-of-Work). One particular instantiation will require state commitments to occupy an entire transaction block (without containing any transactions) and / or to require more stringent hash thresholds on these transaction blocks - thus restricting the rate at which such commitments can be generated.

In order to validate state commitments, we ensure that they have a challenge period of F blocks. Within this challenge period, any node can try to contest this state commitment. If the state commit passes the challenge period, every node on the network assumes that the correct state of shard s at proposer tree level ℓ is committed to $c_{s,\ell}$.

We now describe the state commit challenge protocol, which consists of 5 rounds:

- **Round 1**: State commit is submitted to the entire network by a node eligible to checkpoint (by some Sybil resistant check like PoW). The state commit is then included in the beacon chain. The state commit $\mathcal{C}_{s,\ell}$ consists of the following:
  1. Merkle root $c_{s,\ell}$ of the end state $S_{s,\ell}$ after executing shard s till proposer level ℓ.
  2. List of Merkle roots of states at intermediate blocks $[c_{s,\ell-1}, ..., c_{s,\ell-q}]$ where $c_{s,\ell-q}$ is the previous confirmed state commit.

- **Round 2**: Once the state-commit $\mathcal{C}_{s,\ell}$ is included in the beacon chain, nodes interested in participating in the challenge check if the state commit is correct. If a node finds that the state is not calculated correctly, it publishes $\textit{Disagreement}_1$ $(c_{s,\ell-i})$ where $c_{s,\ell-i}$ is the first intermediate state commit where disagreement occurs. The $\textit{Disagreement}_1$ message consists of an integer i stating the earliest disagreement level. Thus message must be posted on the beacon chain within time F_1 of state-commit $\mathcal{C}_{s,\ell}$.

- **Round 3**: Any honest node (in particular the node that submitted the original state commitment) can reply to the disagreement message with a *Defense* message which commits to state at *all transactions* in the blocks between level $\ell - i$ and level $\ell - i + 1$. For example, let $S_{s,\ell-i,t}$ denote the state of the shard and $c_{s,\ell-i,t}$ denote the corresponding commitment after executing t transactions in the $\ell - i + 1$-th block (here t can be $1, 2, ..., T$ where T is the total number of transactions in the $\ell - i + 1$-th block). The *Defense* message from the checkpointing node comprises of $\{c_{s,\ell-i,1}, ..., c_{s,\ell-i,T}\}$. *Defense* message must be posted within F_d time of the *Disagreement1*.

- **Round 4**: The complainant replies with the *Disagreement2* message which mentions the state commit of the first transaction t the node disagrees with.
  1. Merkle roots of state $[c_{s,\ell-i,t-1}, c_{s,\ell-i,t}]$ where $S_{s,\ell-i,t}$ is the first disagreement to the state roots mentioned in *Defenser* message.
  2. Transaction t of shard s in the proposer block $\ell - i$ and its merkle proof.
  3. Partial state which the transaction $S_{s,\ell-i,t}$ reads/writes. **What is partial state?**

Disagreement2 message must be posted on the beacon chain within time F_2 of the *Defense* message.

- **Round 5**: All nodes run transaction t from the information collected from *Disagreement2* message. If the complainant is correct, the state commit $c_{s,\ell}$ fails and the nodes resort to the previous state commit for shard s.

If at any point during the execution, the complainant message fails to get included in the beacon chain within the requisite time interval, the complainant is assumed to be fraudulent. If at any point of time, the message from the node proposing the checkpoint fails to be published in the beacon chain withing the time window, the state commit is assumed to be fraudulent and the nodes resort to the previous state commit for shard s. We note that the interactive protocol essentially reduces the conflict to a single transaction. We can easily adjust the number of rounds of interaction as well as the resolution of state commit at each round, thus exercising a tradeoff between the size of the initial commitment, number of interaction rounds and the complexity of final verification (which everyone participates in).

We note that there are obvious incentives in place for the protocol: (1) submitting state commits require putting in a deposit, (2) submitting disagreements about state commits (by complaints) require deposits also, (3) at the end of the protocol execution, whichever nodes were correct, gets a reward. Furthermore, we assume that as was done in Truebit [16], there are mechanisms to ensure that forced errors at regular intervals so that potential complainants have incentives to verify all transactions.

Now equipped with the functionality of state commitments, we illustrate in the following how Trifecta exploits it to handle inter-shard transactions. Let us consider a simple inter-shard transaction, spending one unspent coin T_1 in shard 1 to create a new coin T_2 in shard 2, we denote this transaction by $T_1 \to T_2$ for this transaction to be recorded in shard 2 chain, nodes in shard 2 should be able to validate the transaction, but can't do so as they cannot verify the UTXO state of shard 1. We can overcome this problem using state commits of each shard that are published to the blockchain at regular intervals.

Given the existence of state-commitments, the inter-shard transaction $T_1 \to T_2$ can be implemented as follows: Owner of T_1 published a receipt transaction $T_1 \to Tr_1$ in shard 1, the receipt transaction is a transaction whose output cannot be spent. This receipt transaction, once included by a shard block referenced by a beacon block is not part of the state of the Shard 1. **This sentence is confusing.** Let the next confirmed state commitment be $c_{1,\ell}$ of shard 1 at proposal block ℓ that passes the challenge. The client now issues a proof that the transaction $T_1 \to Tr_1$ is contained in the state commitment $c_{1,\ell}$ of shard 1. Nodes in shard 2 (which do not have the state of shard 1) can quickly verify that this is valid and therefore shard 2 now creates a transaction $Tr_1 \to T_2$ in shard 2. The validators in shard 2 can verify the Merkle proof of Tr_1 since they have state $Sc_{s,1,id}$. **Is this correct?** This completes the transaction $T_1 \to T_2$.

3.5 Engineering considerations

Dynamic sharding: We point out that our system can easily support dynamic sharding, where the number of (active) shards varies over time. This is because the shard is simply represented by a shard ID in the transaction block, and it is possible to have many shards (exponential in the size of the shard identifier). Most of these shards may have no activity till activated. We can create a shard-create mechanism that activates new shards, and a shard-close mechanism that closes a

shard. It is easy to create incentive structures to regulate the number of shards - for example, each active shard may be required to pay a certain amount of rent for keeping the shard active.
Heterogeneous sharding: We note that each shard can have its own semantics and its own programming language based on which it interprets the sequence of bits in the shard - for example, one shard can use EVM (Ethereum virtual machine) whereas another shard can use Bitcoin scripting layer. Only nodes that are in that shard need to run this programming layer, and other nodes are simply agreeing on the sequence of bits for this shard. This allows a highly heterogeneous sharding platform, where different shards can be running totally different applications. We note that this is enabled due to the fact that our protocol gives a consensus on a sequence of bits and validity is treated external to the protocol. Furthermore, self-allocation of nodes to shards enables nodes to select the most profitable shards (since nodes will earn shard-specific rewards for mining transaction blocks in a shard), thus creating a market mechanism for node-to-shard allocation, which is very important for heterogeneous sharding.
Light Mining: Our system inverts the relationship between full-nodes and miners. In a standard blockchain system like Bitcoin, miners are required to validate all transactions and hence should be strictly more powerful than full-nodes which only maintain the full state. However, in Trifecta, miners do not need to have the computational resources to actually execute the transactions - only the full-nodes need to have these resources. Miners can mine transaction blocks of a given shard, comprising of a sequence of transactions, without actually validating all of them. Thus it is possible to secure mining rewards for a node that is not necessarily able to execute transactions.

4 Implementation

We have implemented a Trifecta client in about 20,000 lines of code. We describe the architecture of our implementation and highlight several design decisions that are key to its high performance.

4.1 Architecture

Our current prototype runs Bitcoin script [15] on top of our consensus layer. The system architecture is illustrated in Figure 12. Functionally it can be divided into the following three modules:

1. *Block Structure Manager*, which maintains the clients' view of the blockchain, and communicates with peers to exchange new blocks.

2. *Ledger Manager*, which updates the ledger based on the latest blockchain state, executes transactions, and maintains the UTXO set.

3. *Miner*, which mines and assembles new blocks.

The ultimate goal of the Trifecta client is to maintain up-to-date information of the blockchain and the ledger. To this end, it maintains the following four data structures:

1. *Block Structure Database*, residing in persistent storage, stores the graph structure of the blockchain (i.e., the voter blocktrees, proposer blocktree, and transactions blocks referenced) as well as the latest confirmed order of proposer and transaction blocks.

2. *Block Database*, residing in persistent storage, stores every block a client has learned about so far.

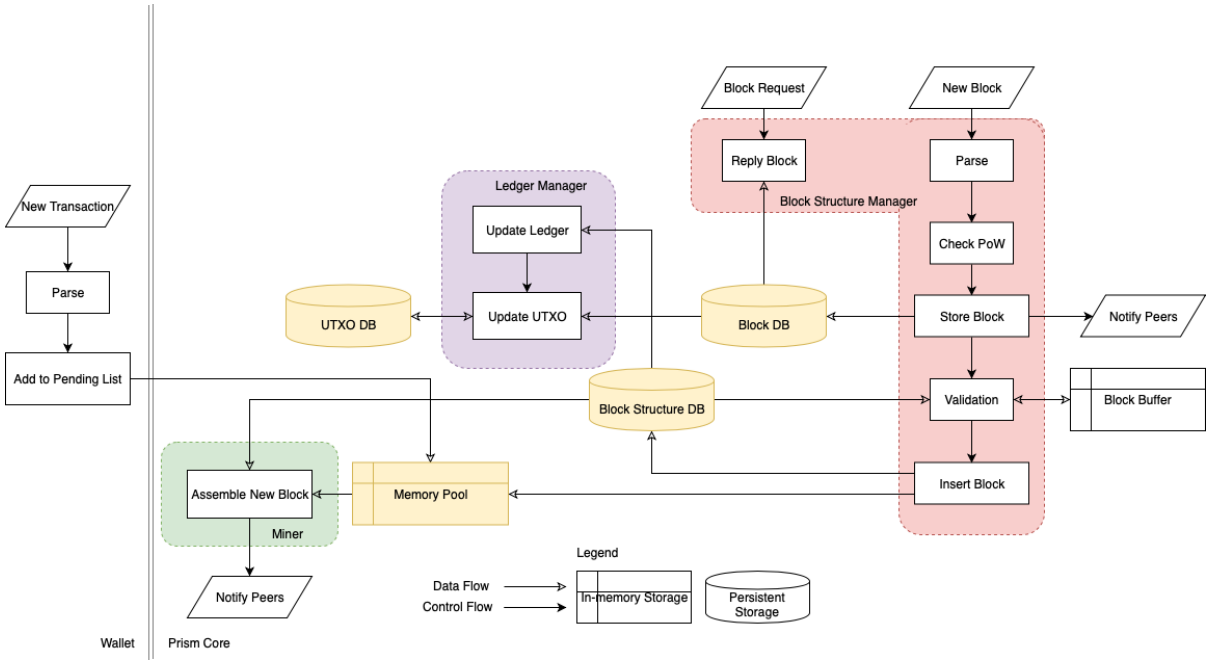


Figure 12: Architecture of Trifecta client.

3. *UTXO Database*, residing in persistent storage, stores the list of all UTXOs, as well as their value and owner.

4. *Memory Pool*, residing in memory, stores the set of unmined transactions.

4.2 Performance Optimizations

The key challenge to implementing the Trifecta client is to handle its high throughput. The client must process blocks at a rate of hundreds of blocks per second, or a throughput of hundreds of Mbps, and confirm transactions at a high rate, around 75,000 tps without sharding and 250,000 tps with sharding in our implementation. To handle the high throughput, our implementation exploits opportunities for parallelism in the protocol and carefully manages race conditions to achieve high concurrency. We now discuss several key performance optimizations.

4.2.1 Asynchronous Ledger Updates

In traditional blockchains like Bitcoin , blocks are mined at a low rate and clients update the ledger each time they receive a new block. However in Trifecta , blocks are mined at a very high rate and a only a small fraction of these blocks — those that change the proposer block leader sequence — lead to changes in the ledger. Therefore trying to update the ledger synchronously for each new block is wasteful and can become a CPU performance bottleneck.

Fortunately, Trifecta does not require synchronous ledger updates to process blocks. In our implementation, the Ledger Manager runs asynchronously with respect to the Block Structure

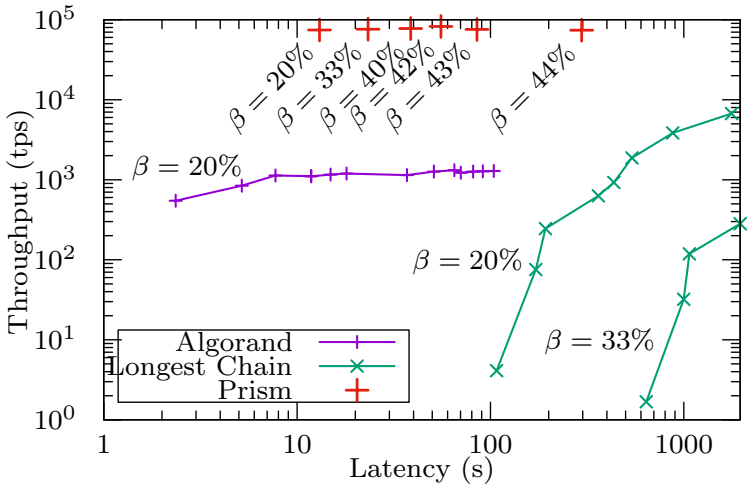


Figure 13: Throughput and confirmation latency of Trifecta , Algorand , and the longest chain protocol on the same testbed. Note that the axes are on log scales. For Algorand and the longest chain protocol, parameters are tuned to span an optimized tradeoff between throughput and latency at a given security level. For Trifecta , throughput and latency are decoupled so one can simultaneously optimize both at one operating point for a given security level.

Manager, to periodically update the ledger.

4.2.2 Parallel Transaction Execution

Executing a transaction involves multiple reads and writes to the UTXO Database to (1) verify the validity of the input coins, (2) delete the input coins, and (3) insert the output coins. If handled sequentially, transaction execution can quickly become the bottleneck of the whole system. Our implementation therefore uses a pool of threads in the Ledger Manager to execute transactions in parallel using *scoreboarding* [17] technique[3].

4.2.3 Functional-Style Design Pattern

Our system must maintain shared state between several modules across both databases and in-memory data structures, creating potential for race conditions. Further, since this state is split between the memory and the database, concurrency primitives provided by the database cannot solve the problem completely. We adopt a functional-style design pattern to define the interfaces for modules and data structures to enable global-lock-free concurrency.

5 Evaluation

Our evaluation answers the following questions:

[3]Despite parallelism, the state update is often the bottleneck for the entire system .

- What is the performance of Trifecta in terms of transaction throughput and confirmation latency, and how does it compare with other protocols?
- How well does Trifecta scale to larger numbers of users?
- How does Trifecta perform with limited resource, and how efficient does it utilize resource?

Testbed: We deploy our Trifecta implementation on 100 Amazon EC2's `c5d.4xlarge` instances connected by a random 4-regular graph topology with propagation delay of 120 ms and 400 Mbps bandwidth. To generate workloads for those experiments, we add a transaction generator in our Trifecta implementation. In our testbed, the main bottleneck is RocksDB and the I/O performance of the underlying SSD, which limits the throughput to about 250,000 tps (resp. 75,000 tps without sharding).

5.1 Scalability

Throughput. As shown in Figure 13, Trifecta is able to maintain the same transaction throughput of around 250,000 tps (resp. 75,000 tps without sharding), regardless of the β chosen. This is because Trifecta decouples throughput from security by using transaction blocks. Algorand and the longest chain protocol do not offer such decoupling, to achieve a higher throughput, one must increase the block size which increases the latency.

Confirmation Latency. The confirmation latency of Trifecta stays below one minute for $\beta \leq 40\%$. At $\beta = 20\%$, Trifecta achieves a latency of 13 seconds, which is comparable to Algorand with similar security guarantees. Compared to the longest chain protocol, Trifecta uses multiple voter chains in parallel to provide security instead of relying on a single chain. So Trifecta requires each vote to be less deep in order to provide the same security guarantee. As a result, Trifecta achieves a substantially lower confirmation latency.

Number of nodes. As shown in Table 14d, both confirmation latency and throughput are constant as the number of clients increases from 100 to 1000.

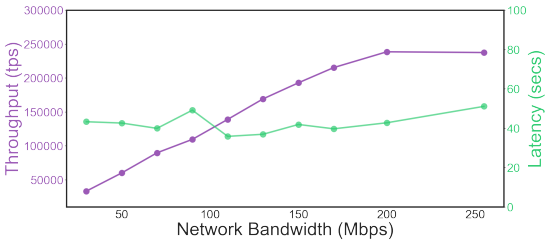
Number of shards. As seen in Figure 14c throughput of Trifecta scales linearly with the number of shards.

Usage			%Bandwidth
Received	Deserialized	Proposer Block	0.05%
		Voter Block	0.21%
		Transaction Block	50.43%
		Messaging	0.43%
	Serialization Overhead		25.80%
Network Headroom			23.08%

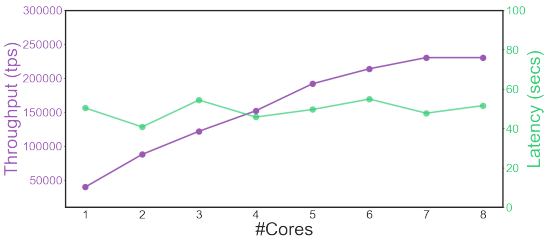
Table 1: Network bandwidth usage breakdown.

Operation		%CPU
Ledger	RocksDB Read/Write	49.5%
	(De)serialization	3.1%
Blockchain	Signature Check	21.7%
	(De)serialization	3.8%
	RocksDB Read/Write	3.9%
	Network I/O	0.6%
Block Assembly		1.5%
Miscellaneous		15%

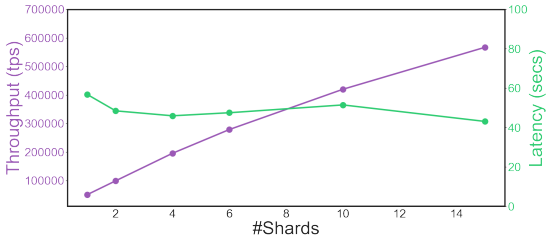
Table 2: CPU usage breakdown.



(a) Performance of Trifecta with different network bandwidth at each client



(b) Performance of Trifecta with different number of CPU cores at each client



(c) Performance of Trifecta with different number of shards

#Nodes	100	300	1000
Throughput (tps)	2.4×10^5	2.5×10^5	2.4×10^5
Latency (s)	40	44	37
Forking	0.119	0.119	0.127

(d) Performance of Trifecta with different network topologies.

Scalability of Trifecta

Bandwidth. Figure 14a shows that the confirmation latency is stable, and the throughput scales proportionally to the available bandwidth. Table 1 provides a breakdown of bandwidth usage.

CPU. Figure 14b shows the throughput scales proportionally to the number of cores and our implementation handles more than 10,000 tps per CPU core. Table 2 provides a breakdown of CPU usage across different components. More than half of CPU cycles are taken by RocksDB. Less than 15% are spent on overhead operations, such as inter-thread communication, synchronization, etc. (categorized as "Miscellaneous" in the table). This suggests that our implementation uses CPU resources efficiently, and further improvements could be made primarily by optimizing the database.

Dashboard. Figures 3 and 4 are the performance snapshot of full-stack implementation of Trifecta .

5.1.1 Overview

Blockchains have inherent incentives that nudge users to produce blocks in compliance with the protocol. The incentive structure in widely used blockchains such as bitcoin and ethereum encourage users to grow the longest-chain using proof-of-work, and these incentive structures have stood the test of time. Designing similar incentives in proof-of-stake blockchains has proven elusive due to a class of attacks called nothing-at-stake, where nodes try to grow blocks on all chains, instead of growing only on the longest chain. In this talk, we identify work-conservation as a key property of proof-of-work blockchains that enables its incentive design. We show how to design a new incentive

mechanism, work virtualization, that mimics the incentive structure of proof-of-work blockchains, thus evading the nothing-at-stake attack. We also show how to fortify this incentive structure against other rational deviations such as selfish mining.

5.1.2 Longest chain PoS protocol

Till now, we have considered the longest chain PoS protocol (PoS-LC) in the adversarial setting, where β is the fraction of stake held by malicious or adversarial users. Another, equally important, consideration in designing a blockchain protocol is the incentive compatibility of the protocol. In this model, rational users may deviate from the prescribed protocol if it is profitable for them to do so.

In this setting, network nodes are categorized into honest and rational. As opposed to the adversarial nodes, a pool of rational nodes, who control α fraction of the stake, try to gain additional block reward by deviating from the protocol. The honest nodes, who control the remaining $1 - \alpha$ fraction of the stake, strictly follow the protocol (PoS-LC) by mining on the tip of the longest chain at any time instance. Recall that given an overall mining rate of f blocks per second on a single parent block, the arrival of blocks for an agent controlling γ fraction of stake follows a Poisson process with rate γf.

5.1.3 Incentive problems with longest chain PoS

A key problem with PoS-LC is that deviating from the longest-chain protocol can be profitable even when the rational controls a small fraction of stake (α is small). This is referred to as the nothing-at-stake attack []. The basic idea is that there is no loss (and potentially some gain) for a rational agent to mine not only on the tip of the longest chain but also on other possible tips of other forks of the blockchain simultaneously. This is in stark contrast to proof-of-work systems where choosing to mine on multiple forks of the blockchain requires splitting the computational power. **We show in the appendix** that for any non-zero fraction of stake controlled by the adversary, nothing-at-stake attack offers a strictly greater reward than honest mining - thus making nothing-at-stake a strictly dominant strategy.

5.1.4 Work virtualization: Adding mining fee

A naive approach to mitigate the nothing-at-stake attack is to say that if the same stake is used to mine on two chains simultaneously, then that stake is burned (slashed). Since no honest miner should do this , this is a reasonable strategy. However, there is a simple way to bypass this strategy by making sure the stake is split into very small portions that the same stake account never wins in adjacent chains but different stake accounts of the miner wins in different chains. Thus this defense can be easily overcome by disaggregating the stake as finely as allowed.

A countermeaure is to set a minimal stake deposit for each node that wants to participate in consensus and slash (burn) the stake of nodes that try to confirm conflicting forks. For this method to work, it not only necessitates a high stake deposit but also a new type of consensus algorithm with *economic finality*[]. Here, we would like to preserve the longest chain consensus algorithm that has been tested in the wild, and try to mimic the incentive structure of proof-of-work as closely as possible without requiring *a priori* commitments of large sums of deposits.

We begin with the observation that any mined block in the proof-of-work setting requires the miner to have paid a certain cost on average (we use M to denote this average mining cost).

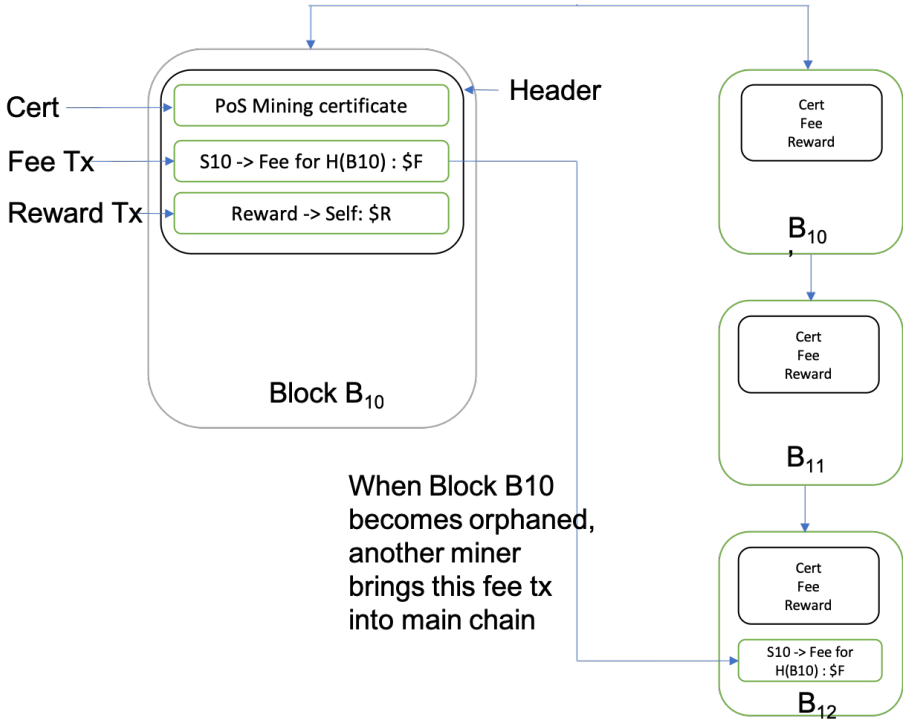


Figure 15: Fees and incentives

This mining cost includes the operational cost of mining, such as energy, as well as the amortized overheads in maintaining a mining rig. However, in a proof-of-stake setting, there is no such cost to be paid - which is the source of the nothing at stake attack. We note that this cost is paid in the hope that the block is included in the main chain and will thereby earn the miner a certain reward. However, in the case that the block does not get included in the main chain, due to forking, the miner obtains no reward and loses this sunk cost.

Our main idea is to emulate this cost in a proof-of-stake system. Whenever a PoS miner gets a mining opportunity, the miner includes a mining fee F along with the block. This mining fee is included in a special transaction, structured as follows: the fee transaction is a transaction whose input is a valid coin and the output goes to a certain fixed address called the "fee address" and there is a special field which includes the hash of the block (the coin is deemed as burnt when it is sent to this address). If the block is included in the main chain, then the fee is charged. Even if the block is not included in the main chain, any other block can pick up this special transaction and include it in its transaction list. Since the fee transaction specifies the hash-of-the-block, it cannot be reused as a fee transaction for a different block but including this transaction in a different block essentially burns the coin thus activating the fee even when the block is not included in the main chain.

Attacks on fee structure The proposed fee structure gets the high level idea of emulating PoW incentives correctly, however, there are a few ways to bypass this mechanism as stated. One possible attack is that a miner that gets potential blocks in multiple different forks use **the same**

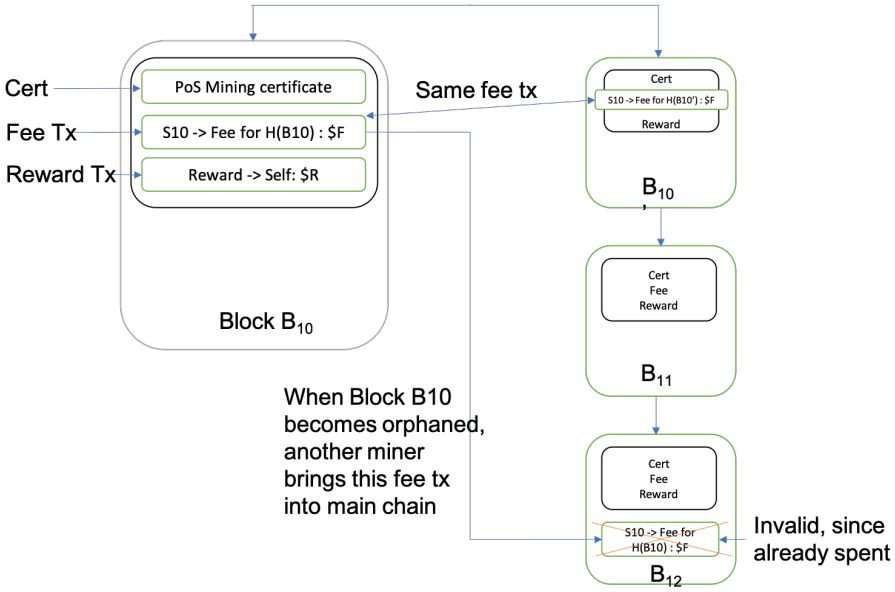


Figure 16: Double-spend fee tx Attack

fee transaction on all the blocks. Since it is the authorized signatory for that transaction, it can sign different versions of the transaction from the same account with different block hashes. Thus, if one block gets finalized, that coin has already been used up in a fee transaction and thus cannot be charged. This attack can be used to effectively evade the fee payable for that transaction.

5.1.5 Defense: Reward Maturation

We note that when the same tax transaction is spent on different chains, there is no way to charge the miner. Since we do not know any further identity of the miner other than the stake that was used to mine, we cannot do anything. However, another level of control that the protocol can exercise is in the reward transaction. Typically, in protocols that utilize the longest-chain mining rule, there is a waiting period for the reward to become available. In our incentive system, we add a further condition to the reward which says that if a proof that the tax-transaction was double-spent is included in any block within that waiting period, then the reward is no longer offered to the miner but we will still charge the first tax transaction to the miner - thus giving a net negative incentive to the miner. Regardless of which fork gets selected, the miner that double-spent the fee transaction ends up with a net negative reward.

Attack-2 and defense We note that we only considered a double-spend attack where the user posts the same fee in blocks in competing chains. An alternative attack happens when a user sends the source of the fee transaction to a different id (on a competing chain). Let us take a simple example, where a user posts a simple

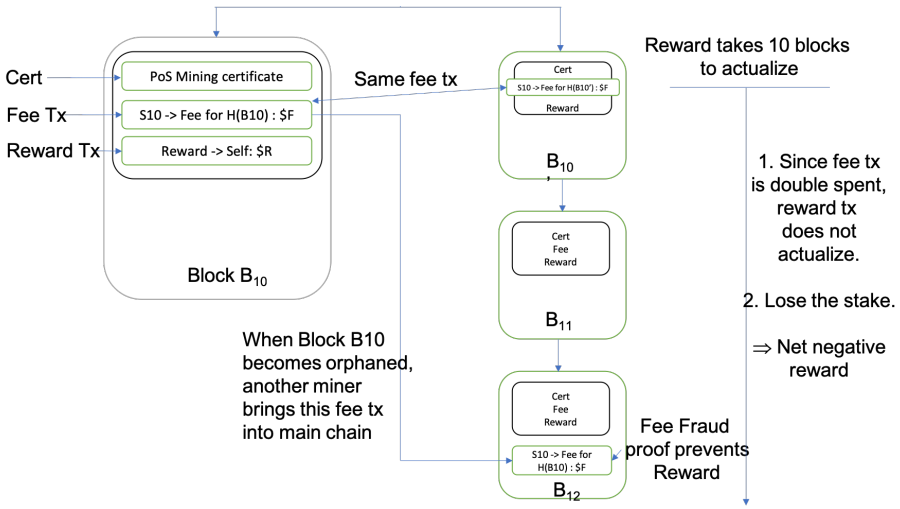


Figure 17: Reward maturation prevents double spend

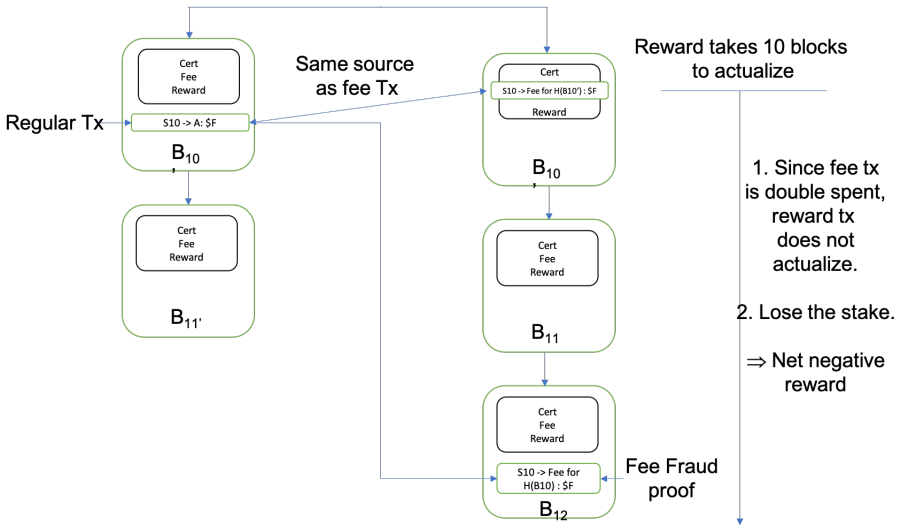


Figure 18: Reward maturation prevents double spend with regular transaction

5.1.6 Work Virtualization: Protocol and Incentives

Now we are ready to summarize the work-virtualization protocol. The following are the constraints of validity.

1. Block is valid only if there is valid certificate, and there is a valid fee payment (relative to the history of the block).
2. The reward of the block actualizes B blocks later unless there is a fee-fraud proof as follows:
   - Any user can post a fraud proof as a special transaction in one of the blocks following the block.
   - A fee-fraud proof basically shows a proof that the fee transaction was double-spent.
   - There are two ways in which a fee transaction can be double-spent: (a) As a fee transaction for a different block. (b) The source of the fee transaction is used as the source of a regular transaction.

5.1.7 Incentive table for Work Virtualization

In proof-of-work systems, the incentives are as follows:

1. *Grow on* 1 *chain*: The expected mining cost is M. If the chain becomes the main chain, then the reward R is collected, i.e., the net reward is $R - M$. If the block does not become part of main chain, then the reward is $-M$.
2. *Grow on i chains*: The expected mining cost is iM. If one of the chain becomes the main chain, then the reward R is collected, i.e., the net reward is $R - iM$. If the none of the i chains become part of main chain, then the reward is $-iM$.
3. *Grow on* 0 *chain*: The net reward is always 0.

In the work virtualization (PoS) system, the following are the possible strategies.

1. *1: Grow on i chains (pay fee on all i)*: The total fee is iF. If one of the chain becomes the main chain, then the reward R is collected, i.e., the net reward is $R - iF$. If the none of the i chains become part of main chain, then the reward is $-iF$ (since all the fee transactions can be posted on these chains).
2. *2:Grow on i chains (use same fee tx on all i)*: The total fee is F. If one of the chain becomes the main chain, then the reward R is forfeited due to fee-fraud proof posted by any honest user but F is charged. If the none of the i chains become part of main chain, then the reward is $-F$.
3. *3:Grow on i chains (use same fee tx on all i), put a competing double spend on all other chains*: The total fee is F. If one of the chain becomes the main chain, then the reward R is forfeited due to fee-fraud proof posted by any honest user but F is charged. If the none of the i chains become part of main chain, then it is not possible to post the fee transaction on any other chain since there is a competing double spend transaction on all other chains.
4. *4: Grow on* 0 *chain*: The net reward is always 0.

We note that in work virtualization, 4 weakly dominates 3 and strongly dominates 2 in reward in all possible executions. Thus a rational user will either use strategy 1 or strategy 4, thus making the system similar to a proof-of-work system (where these strategies remain valid).

Proof-of-work incentive chart

	One of the chains that you grow on becomes MainChain	None of the chains become MainChain
Grow on one chain	R – M	-M
Grow on i chains	R– iM	-iM
Grow on 0 chains	0	0

Work virtualization Incentive Chart

	One of the chains that you grow on becomes MainChain	None of the chains become MainChain
1: Grow on i chains (no double-spend fee)	R – iF	-iF
2: Grow on i chains (double spend fee)	-F	-iS
3: Grow on i chains (double spend fee on these chains, spend fee as reg tx on others)	-F	0
4: Grow on 0 chains	0	0

4 strictly dominates 2

4 weakly dominates 3

Figure 19: Incentive table

5.1.8 Selfish mining: Exaggeration with Private Nothing at Stake

We consider the following selfish mining policy strengthened by the NaS capability of the adversary.

- Define the state using the difference between the length of the private branch and the length of the public branch, denoted by s. This is the lead of the private chain of adversary party.
- When $s \geq 2$, the adversary keeps mining on its private chain. When it observes that the lead drops to 1, it publishes the private chain to override the public chain.
- When the lead $s = 1$, and the adversary observes that the honest party mines a block, it publishes its private chain to match the public chain.
- Zero lead is distinguished into state 0 and state $0'$. State 0 means there is a single tip at the public longest chain. We further divide state 0 into $(0, A)$ and $(0, H)$, where $(0, A)$ means the tip is an adversary block, and $(0, H)$ means the tip is an honest block. State $0'$ means there are two public branches of length 1, one is honest and the other one was published by adversary.
- NaS is done when $s = (0, H)$. Specifically, in addition to mining on the tip of the longest public chain, in this case an honest block, the adversary simultaneously mines on the parent of that block.

5.1.9 Solving Exaggerated Selfish Mining

Fruitchain type incentive structure (naturally offered by Prism) + fee virtualization solves the exaggerated selfish mining problem.

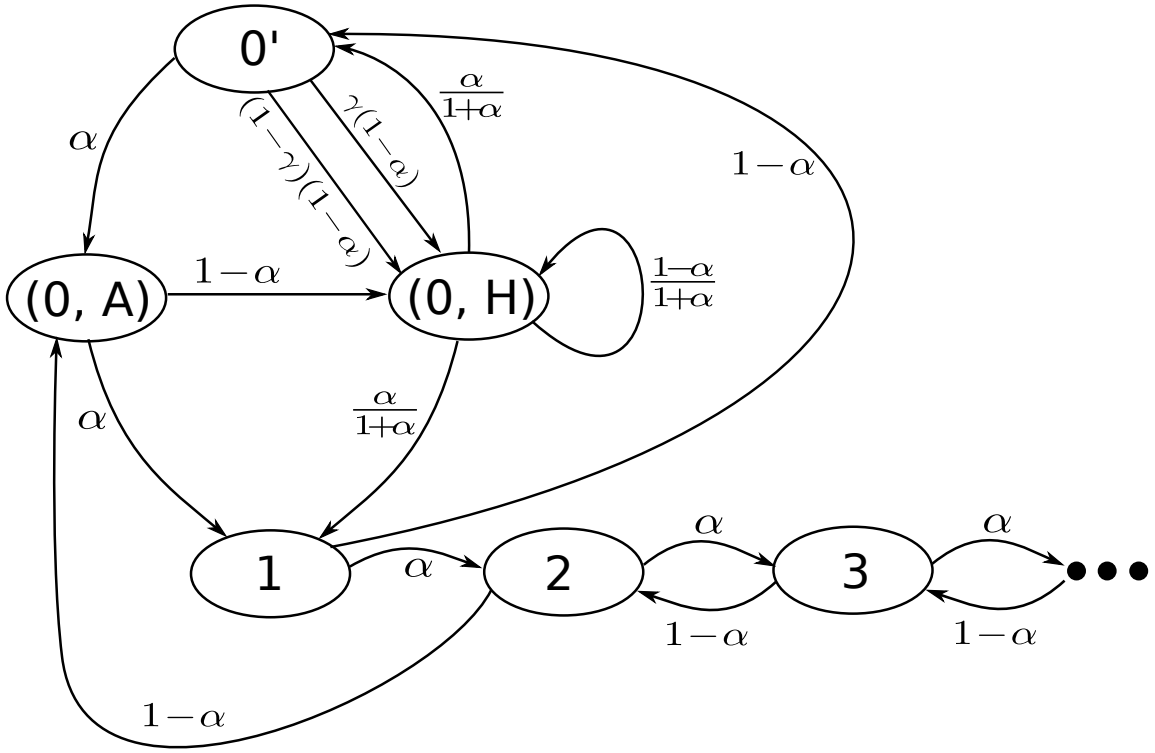


Figure 20: State transition diagram for selfish mining with NaS on honest tip.

References

[1] Bitcoin operating modes. Accessed on Oct. 16, 2019.

[2] The ZILLIQA technical whitepaper. `https://docs.zilliqa.com/whitepaper.pdf`, 2017.

[3] Al-Bassam, M., Sonnino, A., and Buterin, V. Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:1809.09044* (2018).

[4] Bagaria, V., Kannan, S., Tse, D., Fanti, G., and Viswanath, P. Prism : Deconstructing the blockchain to approach physical limits. In *ACM Conference on Computer and Communication Security 2019, see also arXiv:1810.08092* (2019).

[5] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobbler, T., Wei, M., and Davis, J. D. {CORFU}: A shared log design for flash clusters. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012), pp. 1–14.

[6] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 325–340.

[7] Decker, C., and Wattenhofer, R. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings* (Sept 2013), pp. 1–10.

[8] Garay, J., Kiayias, A., and Leonardos, N. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2015), Springer, pp. 281–310.

[9] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., and Ford, B. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 583–598.

[10] Lin, S., and Costello, D. J. *Error control coding*. Pearson, 2004.

[11] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 17–30.

[12] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system.

[13] Orlitsky, A., Viswanathan, K., and Zhang, J. Stopping set distribution of LDPC code ensembles. *IEEE Transactions on Information Theory 51*, 3 (2005), 929–953.

[14] Reed, I. S., and Solomon, G. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[15] Satoshi. Bitcoin utxo script. https://en.bitcoin.it/wiki/Script.

[16] Teutsch, J., and Reitwiessner, C. A scalable verification solution for blockchains. *arXiv preprint arXiv:1908.04756* (2019).

[17] Thornton, J. E. Parallel operation in the control data 6600. In *Proceedings of the October 27-29, 1964, Fall Joint Computer Conference, Part II: Very High Speed Computer Systems* (New York, NY, USA, 1965), AFIPS '64 (Fall, part II), ACM, pp. 33–40.

[18] Wang, X., Kamath, G., Bagaria, V., Kannan, S., Oh, S., Tse, D., and Viswanath, P. Proof-of-stake longest chain protocols revisited. *arXiv* (2019).

[19] Yang, L., Bagaria, V., Wang, G., Alizadeh, M., Fanti, G., Tse, D., , and Viswanath, P. Prism: Scaling bitcoin by 10,000 ×. *arXiv:1909.11261* (2019).

[20] Yu, M., Sahraei, S., Li, S., Avestimehr, S., Kannan, S., and Viswanath, P. Coded Merkle tree: Solving data availability attacks in blockchains. *arXiv preprint arXiv:1910.01247* (2019).

A Pseudocode

Algorithm 1 Prism: Mining

```
1: procedure Main( )
2:     Initialize()
3:     while True do
4:         header, Ppf, Cpf = PowMining()
5:         // Block contains header, parent, content and merkle proofs
6:         if header is a tx block then
7:             block ← ⟨header, txParent, txPool, Ppf, Cpf⟩
8:         else if header is a prop block then
9:             block ← ⟨header, prpParent, unRfTxBkPool, Ppf, Cpf⟩
10:        else if header is a block in voter blocktree i then
11:            block ← ⟨header, vtParent[i], votesOnPrpBks[i], Ppf, Cpf⟩
12:        BroadcastMessage(block)                          ▷ Broadcast to peers
13: procedure Initialize( )                                 ▷ All variables are global
14:    // Blockchain data structure C = (prpTree, vtTree)
15:    prpTree ← genesisP                                   ▷ Proposer Blocktree
16:    for i ← 1 to m do
17:        vtTree[i] ← genesisM_i                           ▷ Voter i blocktree
18:    // Parent blocks to mine on
19:    prpParent ← genesisP                                 ▷ Proposer block to mine on
20:    for i ← 1 to m do
21:        vtParent[i] ← genesisM_i                         ▷ Voter tree i block to mine on
22:    // Block content
23:    txPool ← φ                                           ▷ Tx block content: Txs to add in tx bks
24:    unRfTxBkPool ← φ                                     ▷ Prop bk content1: Unreferred tx bks
25:    unRfPrpBkPool ← φ                                    ▷ Prop bk content2: Unreferred prp bks
26:    for i ← 1 to m do
27:        votesOnPrpBks(i) ← φ                             ▷ Voter tree i bkbk content
28: procedure PowMining( )
29:    while True do
30:        txParent ← prpParent
31:        // Assign content for all block types/trees
32:        for i ← 1 to m do vtContent[i] ← votesOnPrpBks[i]
33:        txContent ← txPool
34:        prContent ← (unRfTxBkPool, unRfPrpBkPool)
35:        // Define parents and content Merkle trees
36:        parentMT ←MerklTree(vtParent, txParent, prpParent)
37:        contentMT ←MerklTree(vtContent, txContent, prContent)
38:        nonce ← RandomString(1^κ)
39:        // Header is similar to Bitcoin
40:        header ← ⟨ parentMT.root, contentMT.root, nonce ⟩
41:        // Sortition into different block types/trees
42:        if Hash(header) ≤ mf_v then                      ▷ Voter block mined
43:            i ← ⌊Hash(header)/f_v⌋ and break             ▷ on tree i
44:        else if mf_v <Hash(header) ≤ mf_v + f_t then
45:            i ← m + 1 and break                          ▷ Tx block mined
46:        else if mf_v + f_t < Hash(header) ≤ mf_v + f_t + f_p then
47:            i ← m + 2 and break                          ▷ Prop block mined
       // Return header along with Merkle proofs
48:    return ⟨header, parentMT.proof(i), contentMT.proof(i)⟩
49:
50: procedure ReceiveBlock(B)                               ▷ Get block from peers
51:    if B is a valid transaction block then
52:        txPool.removeTxFrom(B)
53:        unRfTxBkPool.append(B)
54:    else if B is a valid block on i^th voter tree then
55:        vtTree[i].append(B) and vtTree[i].append(B.ancestors())
56:        if B.chainlen > vtParent[i].chainlen then
57:            vtParent[i] ← B and votesOnPrpBks(i).update(B)
58:    else if B is a valid prop block then
59:        if B.level == prpParent.level+1 then
60:            prpParent ← B
61:            for i ← 1 to m do                            ▷ Add vote on level ℓ on all m trees
62:                votesOnPrpBks(i)[B.level] ← B
63:        else if B.level > prpParent.level+1 then
64:            // Miner doesnt have block at level prpParent.level+1
65:            RequestNetwork(B.parent)
66:        prpTree[B.level].append(B), unRfPrpBkPool.append(B)
67:        unRfTxBkPool.removeTxBkRefsFrom(B)
68:        unRfPrpBkPool.removePrpBkRefsFrom(B) ‘
69: procedure ReceiveTx(tx)
70:    if tx has valid signature then txPool.append(B)
```

Algorithm 2 Prism: Tx confirmation

```
1: procedure IsTxConfirmed(tx)
2:     Π ← φ                                                    ▷ Array of set of proposer blocks
3:     for ℓ ← 1 to prpTree.maxLevel do
4:         votesNdepth ← φ
5:         for i in 1 to m do
6:             votesNdepth[i] ← GetVoteNDepth(i, ℓ)
7:         if IsPropSetConfirmed(votesNdepth) then                          ▷ Refer Def. ??
8:             Π[ℓ] ← GetProposerSet(votesNdepth)                           ▷ Refer Eq. ??
9:         else break
10:    // Ledger list decoding: Check if tx is confirmed in all ledgers
11:    prpBksSeqs ← Π[1] × Π[2] × ··· × Π[ℓ]                                ▷ outer product
12:    for prpBks in prpBksSeqs do
13:        ledger = BuildLedger(prpBks)
14:        if tx is not confirmed in ledger then return False
       return True                          ▷ Return true if tx is confirmed in all ledgers

15: // Return the vote of voter blocktree i at level ℓ and depth of the vote
16: procedure GetVoteNDepth(i, ℓ)
17:    voterMC ← vtTree[i].LongestChain()
18:    for voterBk in voterMC do
19:        for prpBk in voterBk.votes do
20:            if prpBk.level = ℓ then
21:                // Depth is #of children bks of voter bk on main chain
22:                return (prpBk, voterBk.depth)

23: procedure BuildLedger(propBlocks)                          ▷ Input: list of prop blocks
24:    ledger ← []                                             ▷ List of valid transactions
25:    for prpBk in propBlocks do
26:        refPrpBks ← prpBk.getReferredPrpBks()
27:        // Get all directly and indirectly referred transaction blocks.
28:        txBks ← GetOrderedTxBks(prpBk, refPrpBks)
29:        for txBk in txBks do
30:            txs ← txBk.getTxs()                             ▷ Txs are ordered in txBk
31:            for tx in txs do
32:                // Check for double spends and duplicate txs
33:                if tx is valid w.r.t to ledger then ledger.append(tx)
34:    return ledger

35: // Return ordered list of confirmed transactions
36: procedure GetOrderedConfirmedTxs()
37:    L ← φ                                                   ▷ Ordered list of leader blocks
38:    for ℓ ← 1 to prpTree.maxLevel do
39:        votes ← φ                          ▷ Stores votes from all m voter trees on level ℓ
40:        for i in ← 1 to m do
41:            votesNDepth[i] ← GetVotes(i, ℓ)
42:        if IsLeaderConfirmed(votesNDepth) then                   ▷ Refer ??
43:            // Proposer block with maximum votes on level ℓ
44:            L[ℓ] ← GetLeader(votesNDepth)
45:        else break
       return BuildLedger(L)
```