Design and Implementation of a High Performance Blockchain System

by

Lei Yang

B.S., Peking University (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 15, 2020

Certified by .....
Mohammad Alizadeh
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Design and Implementation of a High Performance Blockchain System

by

Lei Yang

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Bitcoin is the first fully-decentralized permissionless blockchain protocol to achieve a high level of security: the ledger it maintains has guaranteed liveness and consistency properties as long as the adversary has less compute power than the honest nodes. However, its throughput is only 7 transactions per second and the confirmation latency can be up to hours. Prism is a new blockchain protocol that is designed to achieve a natural scaling of Bitcoin's performance while maintaining its full security guarantees. In prior work, Prism's security and performance properties have been analyzed theoretically, but the analysis relies on a simple network model and specifies performance bounds up to large constants. Hence, the results cannot predict the protocol's real-world performance.

In this thesis, we present a Bitcoin-like payment system based on the Prism protocol and evaluate it on a network of up to 1000 EC2 virtual machines. Our system achieves a throughput of over $70,000$ transactions per second and a confirmation latency of tens of seconds, validating the prior theoretical results. We introduce several optimizations that allow the system to scale linearly up to 8 CPU cores, and a new algorithm to confirm transactions that is faster and more practical than the original protocol. We also evaluate practical security concerns like the censorship attack, the balancing attack, and spamming, and propose a simple solution that reduces spam traffic by 80% while only adding 5 seconds to the confirmation latency.

Thesis Supervisor: Mohammad Alizadeh
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This research was performed under the supervision of my advisor Professor Mohammad Alizadeh and in collaboration with Vivek Bagaria, Gerui Wang, Professor David Tse, Professor Giulia Fanti, and Professor Pramod Viswanath. I would like to thank them for the help, guidance, and support along the way. It would be impossible to finish this project without these amazing collaborators.

A simple thanks can never express my gratitude towards Mohammad, for being the best advisor and friend one could ever imagine. He provides me with unreserved support and wisdom when I need them the most, and shows me the elegance and joy of system research. I will be forever grateful for the time and energy he pours into me and the project. I feel extremely lucky to be able to work with Mohammad for the past two years, and can not wait for the endeavors with him in the upcoming years.

I would also like to thank Professor David Tse for all the guidance and advice. Discussions with him have always been fruitful. I can never learn enough from his passion for perfection and crave for simple yet powerful mathematical theories. My thanks also go to Professor Hari Balakrishnan, Professor Pramod Viswanath, Professor Giulia Fanti, and Professor Sreeram Kannan for all the help, discussions, and advice. I am lucky to be able to learn from them all.

I would like to thank my labmates and friends, especially Hongzi Mao, Vivek Bagaria, Prateesh Goyal, Lijie Fan, Venkat Arun, InHo Cho, Arjun Balasingam, Vibhaalakshmi Sivaraman, Mehrdad Khani, Parimarjan Negi, Seo Jin Park, and Zeyuan Shang, for the joy, advice, and support they provide. I would like to thank Sheila Marian for her help as I navigate around MIT. I'm also thankful to MIT and CSAIL for providing an amazing environment for me to pursue my passions.

Lastly but most importantly, I thank my girlfriend Xinyue for her unconditioned support, company, and love. Being with her makes me feel courageous and excited about this journey. Finally, I turn to my parents, Ya'nan and Fan. Words can not express my gratitude here. Thank you for raising me, making me who I am, and always being there for me; this thesis is dedicated to you.

Contents

1 Introduction **13**

2 Related Work **17**

3 The Longest Chain Protocol **21**

3.1 Latency Limitation 22

3.2 Throughput Limitation 22

4 Overview of Prism **25**

4.1 Security and Latency 26

4.2 Throughput 28

5 Design **31**

5.1 Notation 31

5.2 Mining 31

5.3 Ledger Formation 34

5.4 Spam Mitigation 36

6 Implementation **39**

6.1 Architecture 39

6.2 Performance Optimizations 42

7 Evaluation **47**

7.1 Throughput and Latency 49

7.2 Scalability 52

7.3 Resource Utilization 52
7.4 Performance Under Active Attack 55

8 Conclusion **59**

A Testbed Design **65**
A.1 Working with an EC2 Cluster 65
A.2 Monitoring the performance 66

B Confirmation Rule **69**

C Parameters Used in the Evaluation **75**

D Block Propagation Delay Distribution **79**

List of Figures

1-1 Throughput and confirmation latency of Prism, Algorand, Bitcoin-NG, and the longest chain protocol on the same testbed. 15

3-1 Depth of confirmation: longest chain vs. Prism. 23

3-2 Reliability as a function of confirmation depth. 23

4-1 Prism factors the blocks into three types of blocks. 26

5-1 Ledger formation of Prism. 35

6-1 Architecture of our Prism client implementation. 40

7-1 Throughput and confirmation latency of Prism, Algorand, Bitcoin-NG, and the longest chain protocol on the same testbed. 51

7-2 Performance of Prism with different network bandwidth at each client. 53

7-3 Performance of Prism with different number of CPU cores at each client. 54

7-4 Effectiveness of random jitter against spam attack. 56

7-5 Performance of Prism under censorship attack. 57

7-6 Performance of Prism under balancing attack. 57

D-1 Block propagation delay in the testbed. 80

List of Tables

7.1 Performance of Prism with different network topologies. 53
7.2 Network bandwidth usage breakdown of Prism. 54
7.3 CPU usage breakdown of our Prism implementation. 55

C.1 Parameters of Prism. 76
C.2 Mining rate of proposer and voter blocks in Prism. 76
C.3 Parameters of Algorand. 76
C.4 Parameters of Bitcoin-NG. 77
C.5 Mining rate in the longest chain protocol. 77

Chapter 1

Introduction

In 2008, Satoshi Nakamoto invented Bitcoin and the concept of *blockchains* [28]. Since then, blockchains have attracted considerable interest for their applications in cross-border payments [19, 20], digital contracts [10, 39, 31] and more. At the heart of Bitcoin and many other blockchain projects is the *Nakamoto longest chain protocol*. It enables an open (permissionless) network of nodes to reach consensus on an ordered log of transactions and is tolerant to Byzantine adversarial attacks with no more than 50% of the compute power in the network. To achieve this high level of security, however, the longest chain protocol severely limits transaction throughput and latency (§3). Bitcoin, for example, supports 3–7 transactions per second and can take hours to confirm a transaction with a high level of reliability [28].

The limitations of the longest chain protocol have led to a flurry of work in recent years on more scalable blockchain consensus protocols (§2 discusses related work). However, until recently, no protocol has been shown to guarantee Bitcoin-level security (up to 50% adversarial power) as well as high throughput and low latency. Prism [6] is the first such protocol. Prism is a Proof-of-Work (PoW) blockchain consensus protocol that is (1) secure against 50% adversarial compute power, (2) can achieve optimal throughput (up to the network communication bandwidth), and (3) can achieve near-optimal confirmation latency (on the order of the network's propagation delay). Prism removes the throughput and latency limitations of the longest chain protocol by systematically decoupling security and throughput in the blockchain (§4). A recent

theoretical paper described the core protocol and analyzed its security properties [6].

While these theoretical results are promising, it is not clear how well they can translate into real-world performance. First, the Prism consensus protocol is much more complex than the longest chain protocol: clients must maintain over 1000 distinct blockchains, which refer to each other to create an intricate directed acyclic graph (DAG) structure, and they must process blocks at very high rates (e.g., 100-1000s of blocks per second at 100s of Mbps) to update these blockchains and confirm transactions. Second, Prism's theoretical analysis relies on several simplifying assumptions (e.g., round-based synchronous communication and a simple network model that ignores queuing delay), and to make the analysis tractable, the performance bounds are specified up to large constants that may not be indicative of real-world performance. Third, Prism's theory focuses on the network as the primary performance bottleneck, but a real high-throughput blockchain system must overcome other potential performance bottlenecks. For example, in addition to achieving consensus on a transaction order, clients must also *execute* transactions and maintain the *state* of the ledger to confirm transaction. Though some academic prototypes ignore transaction execution (e.g., [3, 24]), in practice, it often turns out to be the bottleneck due to its high I/O overhead, c.f., [33], §7.3. Finally, Prism could be vulnerable to spamming, a practical security concern that has not been fully analyzed.

In this thesis, we present the design (§5) and implementation (§6) of a Bitcoin-like system based on the Prism consensus protocol. Our implementation features payments as multi-input-multi-output transactions (payments) similar to pay-to-public-key (P2PK) in Bitcoin and Algorand [17, 1]. We evaluate our system on a testbed of up to 1000 EC2 Virtual Machines connected via an emulated wide area network. Figure 1-1 summarizes the results. Prism consistently achieves a throughput of over 70,000 tps for a range of security levels β denoting the fraction of adversarial compute power. To guarantee a reversal probability of less than 10^{-9}, Prism's latency ranges from 13 seconds against an adversary of power $\beta = 20\%$, to 296 seconds for $\beta = 44\%$. To our knowledge, this makes our system the fastest implementation of a blockchain system with Bitcoin-level security guarantees. Compared to the longest chain protocol, Prism

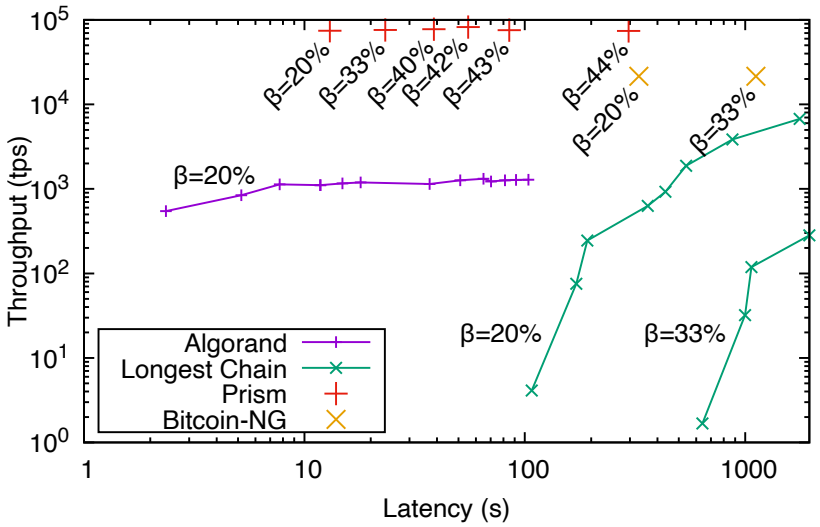


Figure 1-1: Throughput and confirmation latency of Prism, Algorand, Bitcoin-NG, and the longest chain protocol on the same testbed. Note that the axes are on log scales. For Algorand and the longest chain protocol, parameters are tuned to span an optimized tradeoff between throughput and latency at a given security level. For Bitcoin-NG and Prism, throughput and latency are decoupled so one can simultaneously optimize both at one operating point for a given security level. However, the throughput of Bitcoin-NG drops to that of the longest chain protocol under attack, while that of Prism remains high. More details in §2 and §7.1.

provides about 10,000× higher throughput and 1,000× lower latency. Compared to Algorand [17], the state-of-the-art proof-of-stake system, Prism achieves 70× the throughput with about 10 seconds higher latency, and can provide a higher level of security (up to $\beta = 50\%$ vs. $\beta = 33\%$ for Algorand).

We make the following contributions:

- We implement a Prism client in roughly 10,000 lines of `Rust` code, and quantify its performance in extensive experiments on EC2. Our results validate Prism's theoretical results by showing that it can scale both throughput and latency of the longest chain protocol (without compromising security) in a practical setting. Our code is available here [5].

- We propose a new algorithm to confirm transactions that is faster and more practical to implement than the one proposed in the original protocol [6] (see §5.3 and Appendix B).

- Our implementation highlights several performance optimizations, e.g., asynchronous ledger updates, and a scoreboarding technique that enables parallel transaction execution without race conditions (see §6.2). We show that with these careful optimizations, it is possible to alleviate CPU performance bottlenecks and provide linear CPU scaling up to at least 8 cores. At this point, the primary bottleneck for our implementation is the underlying database (RocksDB [4] and I/O to the SSD persistent storage). This suggests that future research on databases optimized for blockchain-specific access patterns could further improve performance.

- We evaluate practical security concerns like censorship attack, balancing attack, and spamming (see §7.4). Additionally, we propose a simple solution to the spamming problem that reduces spam traffic by 80% while only adding 5 seconds to the confirmation delay. Our implementation illustrates that Prism performs well even under these attacks, and makes a stronger case for the practical viability of the system.

The rest of the thesis is organized as follows. In §2 we discuss different scaling approaches taken in blockchains. In §3 we discuss the longest chain protocol and its limitations to motivate the design of the Prism protocol in §4 and §5. We discuss the details of the client implementation with an interface enabling pay-to-public-key transactions in §6. Evaluations are presented in §7 to assess the impact of network resources (bandwidth, topology, propagation delay) and computation resources (memory, CPU) on the overall performance. §8 concludes the thesis.

Chapter 2

Related Work

There are broadly three different approaches to scale the performance of blockchains. First, *on-chain scaling* aims to design consensus protocols with inherently high throughput and low latency. Protocols such as Bitcoin-NG [14], GHOST [36], Algorand [17], OHIE [41] are examples of this approach. Second, in *off-chain scaling*, users establish cryptographically-locked agreements called "payment channels" [13] and send most of the transactions off-chain on those channels. Lightning [32] and Eltoo [12] are examples of this approach. Third, *sharding* approaches conceptually maintain multiple "slow" blockchains that achieve high performance in aggregate. Omniledger [21], Ethereum 2.0 [8], and Monoxide [38] are examples of this approach. These three approaches are orthogonal and can be combined to aggregate their individual performance gains.

Since Prism is an on-chain scaling solution, we compare it with other on-chain solutions. We explicitly exclude protocols with different trust and security assumptions, like Tendermint [22], HotStuff [40], HoneyBadgerBFT [27], SBFT [18], Stellar [25], and Ripple[9], which require clients to pre-configure a set of trusted nodes. These protocols target "permissioned" settings, and they generally scale to significantly fewer number of nodes than the above mentioned permisionless protocols.

Among protocols with similar security assumptions to ours, Bitcoin-NG [14] mines blocks at a low rate similar to the longest chain protocol. In addition, each block's miner continuously adds transactions to the ledger until the next block is mined. This utilizes the capacity of the network between the infrequent mining events, thereby

improving throughput, but latency remains the same as that of the longest-chain protocol. Furthermore, an adversary that adaptively corrupts miners can reduce its throughput to that of the longest chain protocol by censoring the addition of transactions [15]. Prism adopts the idea of decoupling the addition of transactions from the election into the main chain but avoids this adaptive attack. We compare to Bitcoin-NG in §7.1.

DAG-based solutions like GHOST [36], Inclusive [23], and Conflux [24] were designed to operate at high mining rates, and their blocks form a directed acyclic graph (DAG). However, these protocols were later shown to be insecure because they don't guarantee liveness, i.e. the ledger stops to grow, under certain balancing attacks [29]. Spectre[34] and Phantom [35] protocols were built along the ideas in GHOST and Inclusive to defend against the balancing attack, however, they don't provide any formal guarantees. Also, Spectre doesn't give a total ordering and Phantom has a liveness attack [24]. To the best of our knowledge, the GHOST, Inclusive, Spectre and Phantom protocols have no publicly available implementation, and hence we were not able to compare these protocols with Prism in our performance evaluation.

The blockchain structure maintained by Prism is also a DAG, but a structured one with a clear separation of blocks into different types with different functionalities (Figure 4-1). OHIE [41] and Parallel Chains [15] build on these lessons by running many slow, secure longest chains in parallel, which gives high aggregate throughput at the same latency as the longest-chain protocol. To our knowledge, Parallel Chains has not been implemented. In OHIE's latest implementation [3], clients do not maintain the UTXO state of the blockchain and transactions are signed messages without any context, so it is hard to compare with OHIE in our experiments, where all nodes maintain the full UTXO state.

Algorand [17] takes a different approach by adopting a proof of stake consensus protocol and tuning various parameters to maximize the performance. We compare to Algorand in §7.1. Importantly, none of the above protocols simultaneously achieve both high throughput and low latency. Their reported throughputs are all lower than Prism's, and their latencies are all higher than Prism's, except for Algorand which

has a lower latency.

Chapter 3

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*.

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[1] 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

[1] In case of variable proof of work, honest miners mine on the "heaviest chain".

is proven that as long as the adversary has less than 50% hashing power, the ledger has consistency and liveness properties [16]: 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.

3.1 Latency Limitation

A critical attack on the longest chain protocol is the *private double-spend* attack [28], as shown in Figure 3-1(a). Here, an adversary is trying to revert a block 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 [28]. 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 [28]. Figure 3-2 shows the tradeoff between confirmation depth (and therefore latency) and reliability.

3.2 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

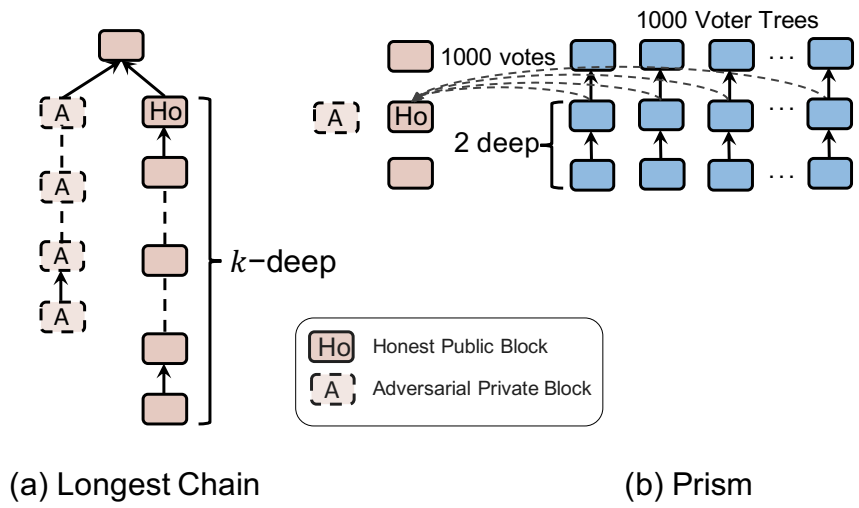


Figure 3-1: Depth of confirmation: longest chain vs. Prism. (a) The longest chain protocol requires a block Ho to be many blocks deep for reliable confirmation, so that an adversary mining in private cannot create a longer chain to reverse block Ho. (b) Prism allows each voter block to be very shallow but relies on many voter chains to increase the reliability.

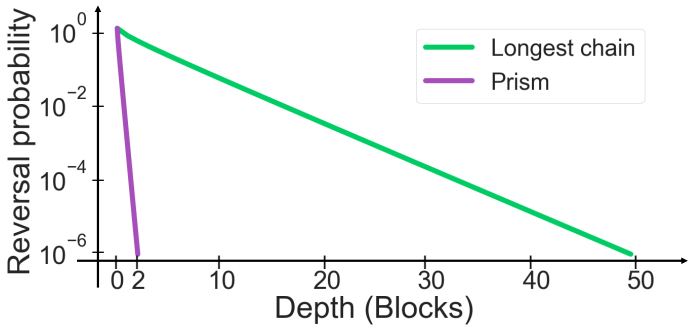


Figure 3-2: Reliability as a function of confirmation depth. The reversal probability of Prism has a factor m improvement over the longest chain protocol in the exponential rate of decrease, where m is the number of voter chains (introduced in §4).

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 [16]. Similarly, increasing the block size B also increases the amount of forking since the network delay Δ increases with the block size [11].

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 [11]. 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 [16]

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

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 $h = 5$. In summary, 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.

Chapter 4

Overview of Prism

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 4-1). 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 be further sortitioned into one of the voter trees. (Mining is described in detail in §5.2).

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*.

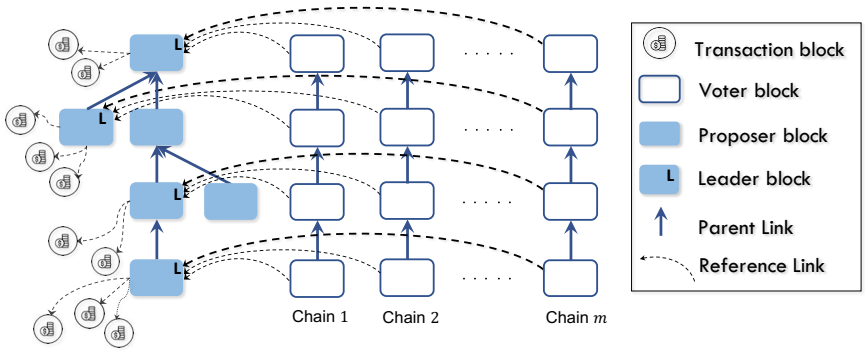


Figure 4-1: Prism: Factorizing the blocks into three types of blocks: proposer blocks, transaction blocks and voter blocks.

There are m voter chains, where $m \gg 1$ is a fixed parameter chosen by the system designer. For example, we choose $m = 1000$ in our experiments. 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. (Ledger formation is explained in detail in §5.3.)

4.1 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 3-1(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 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 [28]. 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, $\epsilon = 0.001$, 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 (§3.1). 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 [6] to obtain the following guarantee on the confirmation latency, regardless of the attack:

Theorem 1 (Latency, Thm. 4.8 [6])**.** *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*

$$Dc_1(\beta) + \frac{Dc_2(\beta)}{m} \log \frac{1}{\epsilon} \quad \textit{seconds}, \tag{4.1}$$

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. Figure 3-2 compares the latency-reliability tradeoffs of Prism and the longest chain protocol. Note that (4.1) is a worst-case latency bound that holds for *all* attacks. In §7.4, we will evaluate the latency of our system under the balancing attack.

4.2 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

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

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 §3.2, the throughput of the longest chain protocol is security-limited, resulting in low network utilization. [6] formally proves that Prism achieves near optimal throughput:

Theorem 2 (Throughput, Thm. 4.4[6])**.** *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.*

Remark on security model: The Prism theory paper [6] analyzed the protocol in a synchronous round-based network model under standard assumptions about the adversary. In particular, the delay for a block of size B was assumed to be equal to $\Delta = \frac{B}{C} + D$, where B/C is the processing delay and D is the propagation delay, and the protocol was assumed to run in rounds where each round is of duration equal to the delay (Δ) corresponding to the largest sized block. The adversarial nodes do not have to follow protocol - they can mine new blocks with any content and anywhere on the blockchain, and unlike honest users, they can keep their mined blocks in private and release them at anytime in the future. However, the adversary cannot modify the content of blocks mined by honest nodes or withhold blocks mined by an honest node from reaching other honest nodes. Refer to §2 of [6] for the full specification of the model. This model does not capture the impact of artifacts like queuing delay or asynchronous communication on performance. Nevertheless our implementation shows that the overall performance characteristics predicted by the theory hold in a practical setting.

Chapter 5

Design

5.1 Notation

Each block $B = (H, C)$ is a tuple containting a header H and content C. As discussed in §4, there are three types of blocks: transaction blocks, proposer blocks, and voter blocks. In all three types, the header $H = (P, n, D)$ is a tuple containing: (1) the hash P of the parent block, (2) a valid PoW nonce n, and (3) a content digest $D = \texttt{Digest}(C)$. We add a superscript to the above notations to denote the type of block being referred. For example, we refer to proposer blocks by B^P, transaction blocks by B^T, and voter blocks by B^V.

5.2 Mining

Miners should not be able to choose *a priori* which type of block they are mining; this is essential for the security of the scheme, since otherwise the adversary could concentrate all of its power on a subset of block trees and overpower them. *Cryptographic sortition* is used to ensure that miners cannot choose which type of block they mine. Nodes simultaneously mine one transaction block, one proposer block, and m voter blocks (one for each tree). Only after a valid proof of work is found does the miner learn if the mined block is a transaction, proposer, or voter block. The mining process has three steps (four including validation):

(1) Superblock generation. When a miner starts mining, it creates a *superblock* that simultaneously contains the parents and contents for all $m+2$ possible sub-blocks (1 transaction sub-block, 1 proposer, and m voter sub-blocks). The parents and contents differ for each type of block. This superblock is updated whenever the miner receives a new network message that changes either the header or the content of any of the sub-blocks.

Transaction sub-block B^T: Transaction blocks do not need a parent block, so $P^T = \emptyset$. The content of a transaction block, C^T, is an ordered list of transactions, drawn from a data structure similar to the Bitcoin `mempool`, except in Bitcoin, `mempool` stores all transactions that have not yet been included in the main chain; in Prism, once a transaction is included in a valid transaction block, it is permanently removed from the `mempool`. This is because the transaction block (hence its contained transactions), is guaranteed to eventually be included in the ledger (§5.3). Upon receiving a new transaction block over the network, the miner should remove the transactions in the new block from its own mempool and transaction block content.

Proposer sub-block B^P: Proposer tree is built in a longest-chain fashion; proposer blocks choose as their parent P^P the tip of the longest chain in the proposer tree. Each proposer block's content, $C^P := (C_1^P, C_2^P)$, is an ordered list of references to other proposer and transaction blocks, where C_1^P is an ordered list of proposer blocks that are neither referenced nor among content of B^P's ancestor block[1], and C_2^P is an ordered list of transaction blocks that are not referenced (directly or indirectly) by any of B^P's ancestors or by any of the proposer blocks in C_1^P. A miner updates content C^P upon receiving a new transaction block or a new proposer block.

Voter sub-block B^{V_i} in the i^{th} voter tree: Voter trees are also built in a longest-chain fashion; the parent of voter block B^{V_i}, P^{V_i}, is the tip of the longest chain in the i^{th} voter tree. The content, C^{V_i}, is a list of references to proposer blocks, or *votes*. Each voter tree's longest chain is allowed to vote at most one proposer block on any level[2] of the proposer tree. Let h denote the last level in the proposer blocktree and ℓ_B

[1] Ancestor blocks are computed by following the chain of links from B^P in the prop. tree.
[2] Level of a proposer block is its distance from the genesis block.

denote the last level voted by B_{V_i}'s ancestors. Then the content of the voter block B^{V_i} is $C^V := [B^P_{\ell'_B+1}, \ldots, B^P_h]$, list of proposer blocks where with some abuse of notation, B^P_ℓ denotes a vote for (pointer to) a proposer block at level ℓ. In words, the voter block contains a list of one vote per unvoted level in the block's ancestors.

By default, nodes will vote for the first proposer block they see at a given level. Notice that the content C^{V_i} is updated if the miner receives either a new proposer block at a previously-unseen level or a new voter block for the i^{th} tree that changes the longest chain of that voter tree. In the former case, the miner adds a vote for that level. In the latter case, the miner updates its parent block P^{V_i} so as to extend the longest chain and also updates the content C^{V_i}. All the contents and parent links are concatenated into a superblock $B = (H, C)$ with header $H = (P := [P^T, P^P, P^{V_1}, \ldots, P^{V_m}], n, D)$ and content $C := [C^T, C^P, C^{V_1}, \ldots, C^{V_m}]$. The content digest D is explained next.

(2) PoW and sortition. Once the superblock is formed, the miner mines by searching for a nonce n such that $Hash(H) \leq q$, where $Hash(\cdot)$ denotes a hash function, and q denotes a difficulty threshold. For a one-way hash function, the miner can do no better than brute-force search, so it cycles through difference values of nonces n until finding one such that $Hash(H) \leq q$. Upon finding a valid nonce, sortition occurs. We divide the numbers from 0 to q into regions corresponding to different block types. For example, $[0, q_T]$ denotes a transaction block, $[q_T + 1, q_P]$ denotes a proposer block, and $[q_P + 1, q]$ denotes voter blocks, split evenly into m regions, one per voter tree. The output region of $Hash(H)$ determines the block type.

(3) Block pruning. Passing around a large superblock after mining would waste unnecessary bandwidth. Hence, to improve space efficiency, instead of using the full concatenated parent block and content lists, only the relevant content is retained after mining and the type of the block is known. For example, a mined proposer block would contain only the proposer parent reference , P^P, and proposer content, C^P; it would *not* store transactions or votes. However, if we do this naively, block validators would not be able to tell if the cryptographic sortition was correctly executed. To address this, we alter our header to contain the following: $H = (\mathsf{MerkleRoot}(P), n, D := \mathsf{MerkleRoot}(C))$, where $\mathsf{MerkleRoot}(\cdot)$ denotes the Merkle root

of a Merkle tree [26] generated from the contained array. In addition to the pruned content and header, we include *sortition proofs*, Merkle proofs attesting to the fact that the block was mined correctly. In our proposer block example, the Merkle proof would include the sibling node for every node in the path from the proposer content C^P to the root $\mathsf{MerkleRoot}(C)$ in the Merkle tree. Hence $\mathsf{MerkleProof}(C)$ (resp. $\mathsf{MerkleProof}(P)$) is an array of size $\log_2(m)$ – a primary source of storage overhead in Prism blocks.

(4) Block validation. Upon receiving a mined Prism block $B = (H, C)$, a validator checks two things. First, it checks that $Hash(H) \leq q$ and that the cryptographic sortition is correct (i.e., that the hash maps to the correct region for the block type). Next, it checks the sortition proof. To do this, it takes content C (resp. parent) in the block, and ensures that the Merkle proof validation gives the content (resp. parent) digest in the header [26].

5.3 Ledger Formation

Prism achieves high throughput in part by mining multiple transaction blocks simultaneously and allowing all of them to contribute to the final ledger. A key consequence is that blocks mined concurrently may contain redundant or conflicting transactions. If Prism were to discard blocks that contain inconsistent transactions, it would needlessly reduce throughput by not confirming the transactions that *are* consistent. To prevent this, Prism separates the process of confirming blocks and forming a ledger. This is a key difference between Prism and many other blockchain protocols. The formation of a ledger in Prism occurs in three steps, as shown in Figure 5-1.

(1) Proposer block confirmation. First, we must confirm a contiguous sequence of leader proposer blocks at each level. Recall that the proposer block with the most votes on level ℓ is defined as the *leader block* at level ℓ, and the sequence of leader blocks for each level of the proposer tree is defined as the *leader sequence*. Once we can guarantee that this leader sequence is permanent for all levels up to some level ℓ with probability at least $1 - \epsilon$, where ϵ is the target reversal probability, we can confirm a leader block sequence. This process is described in more detail below.

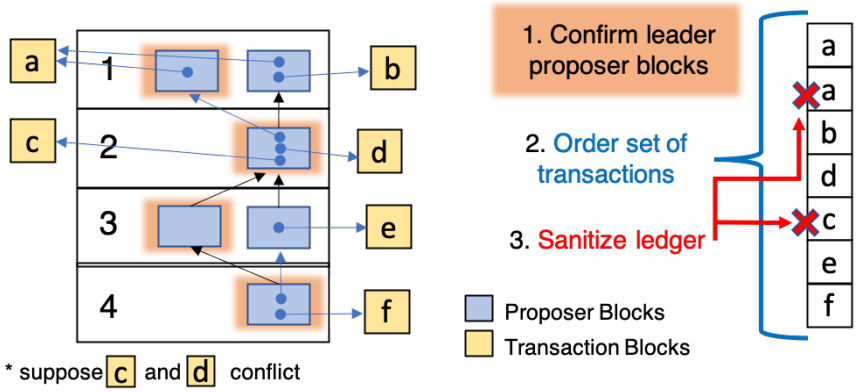


Figure 5-1: Ledger formation has three parts: (1) confirming a leader sequence of proposer blocks; (2) creating a list of transactions; and (3) sanitizing the transaction list for conflicts. In this figure, each transaction block has only one transaction; suppose transactions (c) and (d) are inconsistent (e.g., a double spend). A proposer block's black reference link denotes a parent link. Blue links denote reference links; a proposer block can include reference links to transaction blocks as well as proposer blocks.

(2) Transaction ordering. Given a proposer block leader sequence, we iterate over the sequence and list the referred transaction blocks in the order they are referred. We use L_i to denote the leader at level i. In Figure 5-1, we start with the leader at level 1 L_1, the left proposer block. L_1 refers to only one transaction block containing transaction a, so our ledger starts with a. Next, we consider L_2. It starts by referring to its parent, the right proposer block at level 1. Since that proposer block has not yet been included in the ledger, we include its referred transactions—namely, a and b. L_2 then adds L_1, followed by transaction blocks containing d and c, in that order. Since L_1 was already added to our ledger, we ignore it, but add d and c. This process continues until we reach the end of our leader sequence.

(3) Ledger sanitization. In the previous step, we may have added redundant or conflicting transactions. Hence, we now execute the transaction list in the previously-specified order. Any duplicate or invalid transactions are discarded. In Figure 5-1, we discard the second instance of a (since it's a duplicate), and we discard c (since it conflicts with d).

The key to the above confirmation process is leader proposer block confirmation (step 1). The leader block at a given level ℓ can initially fluctuate when the voter trees

start voting on level ℓ. However, as the voter trees grow, votes on level ℓ are embedded deeper into their respective voter trees, which (probabilistically) prevents the votes from being reverted. Hence, we can confirm the leader block when: (1) a plurality of voter trees have voted for it, and (2) that plurality is guaranteed not to change with probability at least $1 - \epsilon$, where ϵ is a user-selected target reversal probability.

Our confirmation procedure calculates this probability by computing a $(1 - \epsilon)$-confidence interval over the number of votes on each leader block, as well as a hypothetical "private" block that has not yet been released by a hypothetical adversary that controls a fraction β of the hash power. Once the leader block's confidence interval is strictly larger than any of the other candidates' confidence intervals, we can be sure (with probability at least $1 - \epsilon$) that the current leader will remain the leader for all time, so we confirm that proposer block. The details of this confidence interval calculation as well as a brief comparison with the confirmation rule proposed in the original protocol [6] are included in Appendix B.

5.4 Spam Mitigation

In Prism, miners do not validate transactions before including them in blocks. This introduces the possibility of spamming, where an adversary could generate a large number of conflicting transactions and send them to different nodes across the network. The nodes would then mine all of these transactions into blocks, causing miners and validators to waste storage and computational resources.[3] Notice that protocols like the longest chain are not susceptible to this attack because transactions are validated prior to block creation. We propose a simple mechanism to mitigate spamming. Miners validate transactions with respect to their latest ledger state and other unconfirmed transactions, giving the adversary only a small window of network delay to spam the system. This then allows miners to mitigate spamming attacks by adding a random

[3]While a discussion of incentives is beyond the scope of this thesis, it is important to note that fees alone cannot prevent such spamming. Assuming nodes only pay for transactions that make it to into the ledger, the adversary would not be charged for conflicting transactions that get removed during sanitization.

timing jitter prior to mining transactions, thus increasing the chance that a miner can detect that a conflicting transaction is already present in a transaction block, in which case it will choose to not include that transaction. We evaluate the effectiveness of this method in §7.4.

Chapter 6

Implementation

We have implemented a Prism client in about 10,000 lines of Rust code and can be found at [5]. We describe the architecture of our implementation and highlight several design decisions that are key to its high performance.

6.1 Architecture

Our implementation is based on the *unspent transaction output (UTXO)* model, similar to that used by Bitcoin. UTXOs are generated by transactions. A transaction takes a list of UTXOs (*inputs*) and defines a list of new UTXOs (*outputs*). Each UTXO is only allowed to be spent once, and the *state* of the ledger, i.e., the state that results from applying the transactions that have been confirmed up to that point in the ledger, can be represented as a set of UTXOs. Our implementation features a simplified version of Bitcoin's scripting language, processing only pay-to-public-key (P2PK) transactions, similar to that implemented in Algorand [17, 1]. We use Ed25519 [7] for cryptographic signatures and SHA-256 [30] as the hashing algorithm.

The system architecture is illustrated in Figure 6-1. 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.

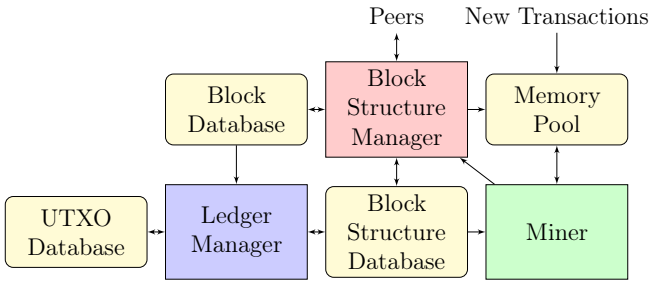


Figure 6-1: Architecture of our Prism client implementation.

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

3. *Miner*, which assembles new blocks.

The ultimate goal of the Prism 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.

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 transactions that have not been mined in any block.

At the core of the **Block Structure Manager** are an *event loop* which sends and receives network messages to/from peers, and a *worker thread pool* which handles those messages. When a new block arrives, the worker thread first checks its proof of work and sortition, according to the rules specified in §5.2, and stores the new block

in the Block Database. [1] It then proceeds to relay the block to peers that have not received it. Next, the worker thread checks whether all blocks referred to by the new block, e.g. its parent, are already present in the database. If not, it buffers the block in an in-memory data structure and defers further processing until all the block's references have been received. Once a block's references have all arrived, the worker performs further validation (e.g., verifying transaction signatures), and finally, the new block is inserted into the Block Structure Database. If the block is a transaction block, the Block Structure Manager also checks the Memory Pool against the transactions included in this new block, and removes any duplicates or conflicting ones.

The **Ledger Manager** is a two-stage pipeline and runs asynchronously with respect to the Block Structure Manager. Its first stage, the *transaction sequencer*, runs in a loop to continuously poll the Block Structure Database and try to confirm new transactions. It starts by updating the list of votes cast on each proposer block. To avoid doing wasteful work, it caches the vote counts and the tips of the voter chains, and on each invocation, it only scans through the new voter blocks. Then, it tries to confirm a leader for each level in the proposer block tree as new votes are cast, according to the rules specified in §5.3. In the case where a leader is selected, it queries the Block Database to retrieve the transaction blocks confirmed by the new leader, and assembles a list of confirmed transactions. The list is passed on to the second stage of the pipeline, the *ledger sanitizer*. This stage maintains a pool of worker threads that executes the confirmed transactions in parallel. Specifically, a worker thread queries the UTXO Database to confirm that all inputs of the transaction are present; their owners match the signatures of the transaction; and the total value of the inputs is no less than that of the outputs. If execution succeeds, the outputs of the transaction are inserted into the UTXO Database, and the inputs are removed.

The **Miner** module assembles new blocks according to the mining procedure described in §5.2. It is implemented as a busy-spinning loop. At the start of each round, it polls the Block Structure Database and the Memory Pool to update the block it is mining. It also implements the spam mitigation mechanism described in

[1] Checking proof of work at the earliest opportunity reduces the risk of DDoS attacks.

§5.4. Like other academic implementations of PoW systems [41, 24], our miner does not actually compute hashes for the proof of work, and instead simulates mining by waiting for an exponentially-distributed random delay. Solving the PoW puzzle in our experiments would waste energy for no reason, and in practice, PoW will happen primarily on dedicated hardware, e.g., application-specific integrated circuits (ASICs). So the cost of mining will not contribute to the computational bottlenecks of the consensus protocol.

The three databases residing in the persistent storage are all built on RocksDB [4], a high-performance key-value storage engine. We tuned the following RocksDB parameters to optimize its performance: replacing B-trees with hash tables as the index; adding bloom filters; adding a 512 MB LRU cache; and increasing the size of the write buffer to 32 MB to sustain temporary large writes.

6.2 Performance Optimizations

The key challenge to implementing the Prism 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, exceeding 70,000 tps 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.

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 Prism, 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, Prism does not require synchronous ledger updates to process blocks. Since Prism allows conflicting or duplicate transactions to appear in the ledger and

performs sanitization later (§5.3), the client need not update the ledger for each new block. Therefore, in our implementation, the Ledger Manager runs asynchronously with respect to the Block Structure Manager, to periodically update the ledger. Most blockchain protocols (e.g., Bitcoin, Algorand, and Bitcoin-NG) require that miners validate a block against the current ledger prior to mining it, and therefore cannot benefit from asynchronous ledger updates. For example, in Bitcoin's current specification, when a miner mines a block B, it implicitly also certifies a ledger L formed by tracing the blockchain from the genesis block to block B. A Bitcoin client must therefore verify that a block B that it receives does not contain transactions conflicting with the ledger L, and hence must update the ledger synchronously for each block. In principle, Bitcoin could perform *post hoc* sanitization like Prism; however, due to long block times relative to transaction verification, doing so would not improve performance.

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.[2] However, naively executing all transactions in parallel is problematic, because semantically the transactions in the ledger form an order, and must be executed strictly in this order to get to the correct final state (i.e., UTXO set). For example, suppose transactions T and T' both use UTXO u as input, and T appears first in the ledger. In this case, T' should fail, since it tries to reuse u when it has already been spent by T. However, if T and T' are executed in parallel, race condition could happen where the inputs of T' are checked before T deletes u from the UTXO Database, allowing T' to execute.

To solve this problem, we borrow the *scoreboarding* [37] technique long used in processor design. A CPU employing this method schedules multiple instructions to be executed out-of-order, if doing so will not cause conflicts such as writing to the

[2]Despite parallelism, the UTXO database is the bottleneck for the entire system (§7.3).

same register. Transactions and CPU instructions are alike, in the sense that they both need to be executed in the correct order to produce correct results, only that transactions read and write UTXOs while CPU instructions read and write CPU registers. In the Ledger Manager, a batch of transactions are first passed through a controller thread before being dispatched to one of the idle workers in the thread pool for execution. The controller thread keeps track of the inputs and outputs of the transactions in the batch on the scoreboard (an in-memory hash table). Before scheduling a new transaction for execution, it checks that none of its inputs or outputs are present on the scoreboard. In this way, all worker threads are able to execute in parallel without any synchronization.

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 RocksDB cannot solve the problem completely. For example, to update the ledger, the Ledger Manager needs to fetch the tips of the voter chains from the memory and the votes from the Block Structure Database, and they must be in sync. Locking both states with a global mutex is a straightforward solution; however, such coarse locks significantly hurt performance.

We adopt a functional-style design pattern to define the interfaces for modules and data structures. Specifically, we abstract each module into a function that owns no shared state. Instead, state is passed explicitly between modules as inputs and outputs. For example, the functionality of the Ledger Manager can be abstracted as `UpdateLedger`$(V, V') \to \Delta T$, where V and V' are the previous and current voter chain tips, and ΔT are the transactions confirmed by votes between V and V'. Then, we design the database schema to support such functions. For example, the Block Structure Database supports the query `VoteDiff`$(V, V') \to \Delta\text{Votes}$, where ΔVotes are the added and removed votes when the voter chains evolve from V to V'. In this way, function `UpdateLedger` can invoke `VoteDiff` to update the votes and confirm new transactions with no need for explicit synchronization, because each function guarantees the correctness of its output with respect to its input. Functional-style

design has broader benefits than enabling global-lock-free concurrency. One example is it facilitates bootstrapping (discussed in §8), where a client needs the ledger formed by leader blocks until a certain level. Another example is reverting to a previous version of the ledgers. Such queries are easily supported in our model by calling the above update ledger function.

No Transaction Broadcasting. In most traditional blockchains, clients exchange pending transactions in their memory pools with peers. This incurs extra network usage, because each transaction will be broadcast twice: first as a pending transaction, and then again as part of a block. At the throughput in which Prism operates, such overhead becomes even more significant.

Our implementation does not broadcast pending transactions, because it is *unnecessary* in Prism. In traditional blockchains like Bitcoin and Ethereum, the whole network mines a block every tens of seconds or even few minutes. Since we cannot predict who will mine the next block, exchanging pending transactions is necessary, so that they get included in the next block regardless of who ends up mining it. In contrast, Prism generates hundreds of transaction blocks every second. This elevated block rate means that any individual miner is likely to mine a transaction block in time comparable to the delay associated with broadcasting a transaction to the rest of the network (i.e., seconds). Hence, unlike other blockchain protocols, there is little benefit for a Prism client to broadcast its transactions. Non-mining clients can transmit their transactions to one or more miners for redundancy; however, those miners do not need to relay those transactions to peers.

Chapter 7

Evaluation

Our evaluation answers the following questions:

- What is the performance of Prism in terms of transaction throughput and confirmation latency, and how does it compare with other protocols? (§7.1)
- How well does Prism scale to larger numbers of users? (§7.2)
- How does Prism perform with limited resource, and how efficient does it utilize resource? (§7.3)
- How does Prism perform when under attack? (§7.4)

Schemes compared: We compare Prism with Algorand, Bitcoin-NG, and the longest chain protocol. For Bitcoin-NG and the longest chain protocol, we modify and use our Prism codebase to enable a fair comparison of the protocols. For Algorand, we use the official open-source implementation [1] written in Golang. Note that this implementation is different from the one evaluated in [17]. Therefore, we do not expect to reproduce the results in [17].

Testbed: We deploy our Prism implementation on Amazon EC2's `c5d.4xlarge` instances with 16 CPU cores, 16 GB of RAM, 400 GB of NVMe SSD, and a 10 Gbps network interface. Each instance hosts one Prism client. By default, we use 100 instances and connect them into a random 4-regular graph topology. To emulate a

wide-area network, we introduce a propagation delay of 120 ms on each link to match the typical delay between two distant cities [2], and a rate limiter of 400 Mbps for ingress and egress traffic respectively on each instance. We also evaluate several other network topologies (with up 1000 instances) and per-instance bandwidth limits. More details on the testbed are in §A

To generate workloads for those experiments, we add a transaction generator in our testbed which continuously creates transactions at an adjustable rate. In our Prism implementation, the main bottleneck is RocksDB and the I/O performance of the underlying SSD, which limits the throughput to about $80,000$ tps. We cap transaction generation rate to 75,000 tps to avoid hitting this bottleneck.

Performance tuning and security: All protocols in the experiments have design parameters, and we tried our best to tune these parameters for performance and security. For Prism, we calculate the optimal mining rate f for proposer and voter blocks to achieve the best confirmation latency, given the adversarial ratio β and desired confirmation confidence ϵ. We cap the size of transaction blocks to be 40 KB, and set the mining rate for transaction blocks such that they support $80,000$ tps. Unless otherwise stated, we turn off the spam mitigation mechanism in Prism (we evaluate its effectiveness in §7.4). To ensure security, we calculate the expected *forking rate* α, i.e. fraction of blocks not on the main chain, given f and the block propagation delay Δ. We compare α against the forking rate actually measured in each experiment, to ensure that the system has met the target security level. We follow the same process for Bitcoin-NG and the longest chain protocol. For Algorand, we adopt the default security parameters set in its production implementation. Then we hand-tune its latency parameters λ and Λ. Specifically, we reduce λ and Λ until a round times out, and use the settings that yield the best confirmation latency. For Prism, we target a confirmation confidence, ϵ, in the order of 10^{-9}. For Bitcoin-NG and the longest chain protocol, we target ϵ in the order of 10^{-5}. For Algorand, the blockchain halts with a probability in the order of 10^{-9}.

7.1 Throughput and Latency

In this experiment, we measure the transaction throughput and confirmation latency of Prism at different adversarial ratio β, and compare that with Algorand, Bitcoin-NG and the longest chain protocol. For Algorand, we use its default setting of security parameters, which targets $\beta = 20\%$.[1] For Bitcoin-NG and the longest chain protocol, we experiment with two adversarial ratios: $\beta = 20\%$ and $\beta = 33\%$. In both Algorand and the longest chain protocol, there is tradeoff between throughput and confirmation latency by choosing different block sizes. We explore this tradeoff and present it in a curve. For Algorand, we try block sizes between 300 KB to 32 MB. For the longest chain protocol, we try block sizes between 1.7 KB to 33.6 MB. The parameters used in this experiment are available in Appendix C. All four protocols are deployed on the same hardware and network topology as described above. We run each experiment for a minimum of 10 minutes and report the average transaction throughput and latency. The results are shown in Figure 7-1.

Throughput: As shown in Fig. 7-1, Prism is able to maintain the same transaction throughput of around $75,000$ tps regardless of the β chosen. This is because Prism decouples throughput from security by using transaction blocks. In this way, Prism is able to maintain the mining rate for transaction blocks to sustain a constant throughput, while changing the mining rate for other types of blocks to achieve the desired β. Bitcoin-NG offers a similar decoupling by entitling the miner of the latest key block to frequently produce micro blocks containing transactions. Algorand and the longest chain protocol do not offer such decoupling, so one must increase the block size in order to achieve a higher throughput. In such case, the confirmation latency increases, as demonstrated by the tradeoff curves in Figure 7-1, to accommodate for the higher block propagation delay induced by larger blocks. For the longest chain protocol, its throughput limit has been discussed in §3.2. For Algorand, we observe its throughput increases marginally with block size, but does not exceed 1300 tps. The reason is that Algorand only commits one block every round. So at any moment,

[1] The maximum possible security level for Algorand is $\beta = 33\%$, but its latency is expected to increase substantially as β approaches 33% [17].

unlike Prism, Algorand only has one block propagating in the network, causing low bandwidth utilization. For Bitcoin-NG, we observed a peak throughput of 21,530 tps. The reason is that, unlike Prism, in Bitcoin-NG only a single node (the leader) commits transactions at a time. This results in the network becoming a bottleneck; once throughput exceeds about 20,000 tps, we observed that the block propagation delay increases significantly for Bitcoin-NG.[2]

Is Consensus the Throughput Bottleneck? A blockchain client has two roles: (1) it participates in the consensus protocol (the *Block Structure Manager* and the *Miner* in our implementation); (2) it executes transactions confirmed by the consensus protocol and updates the ledger (the *Ledger Manager* in our implementation). The throughput can be bottlenecked by either of these stages and therefore we ask: Is the throughput limited by the consensus protocol, or the ledger updates? To answer this question, we measure the maximal throughput when no consensus protocol is involved, i.e. we start one client of each protocol and test how fast each client can execute transactions and update the ledger. For our Prism, Bitcoin-NG and longest chain client, the limit is around $80,000$ tps. For Algorand, the limit is around $4,800$ tps. From Fig. 7-1 we see that Bitcoin, Bitcoin-NG, and Algorand have throughput much lower than these limits, and thus are bottlenecked by the consensus protocols. However, in case of Prism, its throughput is very close to the limit, and hence it is bottlenecked by the ledger updates.

Confirmation Latency: The confirmation latency of Prism stays below one minute for $\beta \leq 40\%$. At $\beta = 20\%$, Prism achieves a latency of 13 seconds, and for similar security guarantees Algorand achieves latency of 2 seconds. Compared to the longest chain protocol, Prism uses multiple voter chains in parallel (1000 chains in our experiments) to provide security instead of relying on a single chain. So Prism requires each vote to be less deep in order to provide the same security guarantee. As a result, Prism achieves a substantially lower confirmation latency. For example, for $\beta = 33\%$, the confirmation latency for Prism is 23 seconds, compared to 639 seconds at the

[2]Note also that Bitcoin-NG is susceptible to an adaptive attack that censors the chosen leader and can reduce throughput substantially [15].

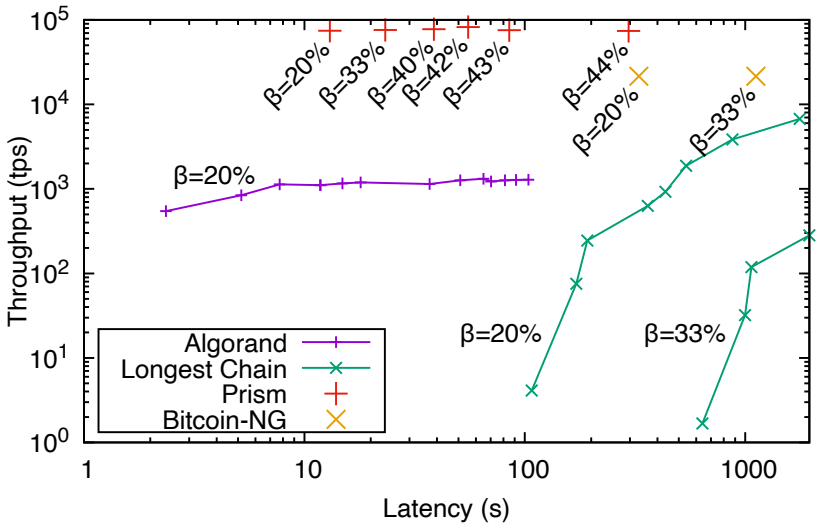


Figure 7-1: Throughput and confirmation latency of Prism, Algorand, Bitcoin-NG, and the longest chain protocol on the same testbed. Note that the axes are on log scales. For Algorand and the longest chain protocol, parameters are tuned to span an optimized tradeoff between throughput and latency at a given security level. For Bitcoin-NG and Prism, throughput and latency are decoupled so one can simultaneously optimize both at one operating point for a given security level. However, the throughput of Bitcoin-NG drops to that of the longest chain protocol under attack, while that of Prism remains high.

lowest throughput point for the longest chain protocol. As we increase the block size for the longest chain protocol, its confirmation latency increases to 1956 seconds at a throughput of 282 tps. The gap between Prism and the longest chain protocol increases for higher β. For example, for Prism the confirmation latency increases from 13 seconds to 23 seconds as β increases from 20% to 33%. For the longest chain protocol, the same change in β causes the latency to increase by more than 800 seconds. Bitcoin-NG exhibits similar confirmation latency as the longest chain protocol for the same value of β, since it applies the same k-deep rule as the longest chain protocol for key blocks to confirm transactions, and key blocks must be mined slowly to avoid frequent leader changes.

7.2 Scalability

In this experiment, we evaluate Prism's ability to scale to a large number of users. For each client, we use the same network and hardware configuration as in other experiments, and target an adversarial ratio $\beta = 40\%$. The results are shown in Table 7.1.

First, we increase the number of clients while keeping the topology a random 4-regular graph, i.e., each client always connects to four random peers. In this case, the network diameter grows as the topology becomes larger, causing the block propagation delay to increase and the confirmation latency to increase correspondingly. Note that the transaction throughput is not affected[3] because in Prism the mining rate for transaction blocks is decoupled from that of the other types of blocks. Then, we explore the case where clients connect to more peers as the topology grows larger, so that the diameter of the network stays the same. As shown in the results, both confirmation latency and throughput are constant as the number of clients increases from 100 to 1000.

In all cases, the forking rate stays stable and is under 0.13, proving that the system is secure for $\beta = 40\%$. This suggests that Prism is able to scale to a large number of users, as long as the underlying peer-to-peer network provides a reasonable block propagation delay. We also provide the distributions of block propagation delay in each topology in Appendix D.

7.3 Resource Utilization

In this experiment, we evaluate the resource utilization of our Prism implementation, and how it performs with limited network bandwidth and CPU resources.

Network Bandwidth: Figure 7-2 shows the throughput and confirmation latency of Prism as we throttle the bandwidth at each client. Results show that the confirmation

[3]In the results, the throughput increases as we increase the network size. This is because of an artifact in our testbed which causes slightly more transactions to be generated when there are more nodes in the network.

Table 7.1: Performance of Prism with different network topologies.

Property	#Nodes	100	300	1000
Degree = 4	Diameter	5	7	9
	Throughput (tps)	7.2×10^4	7.4×10^4	7.4×10^4
	Latency (s)	40	58	67
	Forking	0.119	0.117	0.112
Diameter = 5	Degree	4	6	8
	Throughput (tps)	7.2×10^4	7.9×10^4	7.9×10^4
	Latency (s)	40	44	37
	Forking	0.119	0.119	0.127

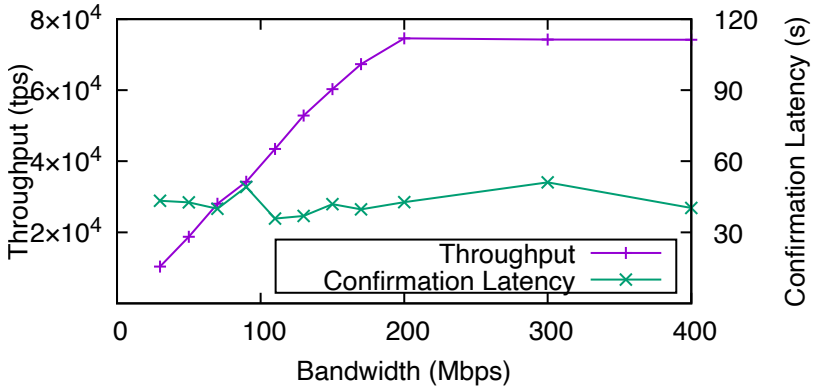


Figure 7-2: Performance of Prism with different network bandwidth at each client. The in-memory size of a transaction is 168 bytes.

latency is stable, and the throughput scales proportionally to the available bandwidth. The throughput stops to grow when the bandwidth is higher than 200 Mbps, because the transaction generation rate is capped at 75,000 tps, which is near the bottleneck caused by RocksDB.

Table 7.2 provides a breakdown of bandwidth usage. Our implementation is able to process transaction data at a throughput about 50% of the available bandwidth. Further improvements could be made by using more efficient data serialization schemes and optimizing the underlying P2P network.

CPU: Figure 7-3 shows the throughput of Prism as we change the number of CPU cores for each client. The throughput scales proportionally to the number of cores, and stops to grow after 7 cores because the transaction generation rate is capped. This shows that our implementation handles more than 10,000 tps per CPU core, and

Table 7.2: Network bandwidth usage breakdown of Prism measured on a 200 Mbps interface. Network Headroom is the unused bandwidth necessary for the block propagation delay to stay stable. Serialization overhead is wasted space when serializing in-memory objects for network transmission. Messaging stands for non-block messages.

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%

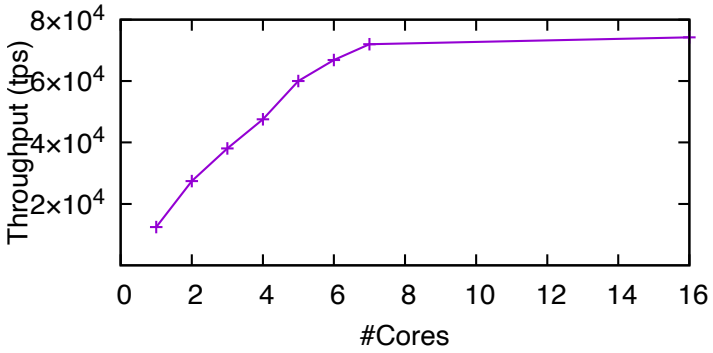


Figure 7-3: Performance of Prism with different number of CPU cores at each client.

the parallelization techniques discussed in §6 are effective.

Table 7.3 provides a breakdown of CPU usage across different components. More than half of CPU cycles are taken by RocksDB for which we only perform basic tuning. 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.

While we chose mid-end AWS EC2 instances for experiments, our results show that Prism does not inherently require powerful machines or waste resources.[4] On the contrary, its high resource efficiency and scalability that we demonstrate in this experiment makes Prism suitable for applications with different requirements.

[4]For example, a laptop with 8 cores, 16 GB RAM, and 400 GB of NVMe-based SSD would cost under $3,000 today and could easily run Prism.

Table 7.3: CPU usage breakdown of our Prism implementation.

Operation		%CPU
Ledger	RocksDB Read/Write	49.5%
	(De)serialization	3.1%
	Miscellaneous	8.9%
Blockchain	Signature Check	21.7%
	(De)serialization	3.8%
	RocksDB Read/Write	3.9%
	Network I/O	0.6%
	Miscellaneous	5.5%
Block Assembly		1.5%
Transaction Generation		0.7%
Miscellaneous		0.8%

7.4 Performance Under Active Attack

In the following experiments, we evaluate how Prism performs in the presence of active attacks. Specifically, we consider three types of attacks: *spamming*, *censorship*, and *balancing* attacks. Spamming and censorship attacks aim to reduce network throughput, while balancing attacks aim to increase confirmation latency. In these experiments we configure Prism to tolerate a maximum adversarial ratio $\beta = 40\%$.

Spamming Attack. Recall that in a spamming attack, attackers send conflicting transactions to different nodes across the network. As described in §5.4, miners can mitigate such attack by adding a random timing jitter to each transaction. In this experiment, we set up 100 miners as victims and connect them according to the same topology as in other experiments. Then for each miner we start a local process that generates a transaction every 100 ms. We synchronize those processes across the network so that each miner receives the same transaction at the same time, with a time synchronization error of several ms due to the Network Time Protocol. To defend against the attack, miners add a uniform random delay before including a transaction into the next transaction block. We let each attack to last for 50 seconds, and measure the fraction of spam transactions that end up in transaction blocks. Fig. 7-4 shows that adding a random jitter of at most 5 seconds can reduce the spam traffic by about 80%. We point out that miners can extend this method by monitoring the reputation

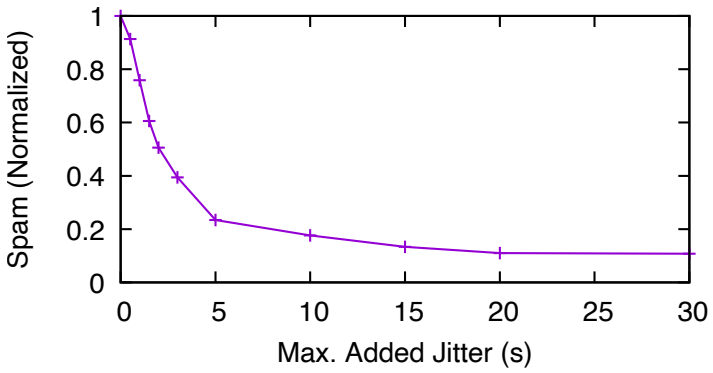


Figure 7-4: Effectiveness of random jitter in defending against spam attack. Jitters follow uniform distributions and we report the maximum jitter that we add. Spam traffic amount is normalized to the case when no jitter is added.

of clients by IP address and public key, and penalizing clients with high spam rate with longer jitter.

Censorship Attack. In a censorship attack, malicious clients mine and broadcast empty transaction blocks and proposer blocks. Censorship attack does not threaten the security of Prism, but it reduces the system throughput because a portion of blocks are now "useless" since they do not contain any data. As Figure 7-5 shows, during a censorship attack, the transaction throughput reduces proportionally to the percentage of adversarial users. Theoretically, censorship attack could also affect the confirmation latency, because it could take longer for a transaction block to be referred to if some proposer blocks are empty. However, since a proposer block is mined roughly every 10 seconds, the impact on latency is nominal. Our results shows that the confirmation latency stays stable as we increase the adversarial ratio from 0% to 25%.

Balancing Attack. In a balancing attack, attackers try to increase the confirmation latency of the system by waiting for the event when multiple proposer blocks appear on the same level, and then balancing the votes among them. Normally, when multiple proposer blocks appear on one level, every client votes for the proposer block with the most votes, so the system quickly converges with the vast majority of voter chains voting for one proposer block. During a balancing attack, however, the attacker votes on the proposer blocks with second most votes to slow down such convergence, causing

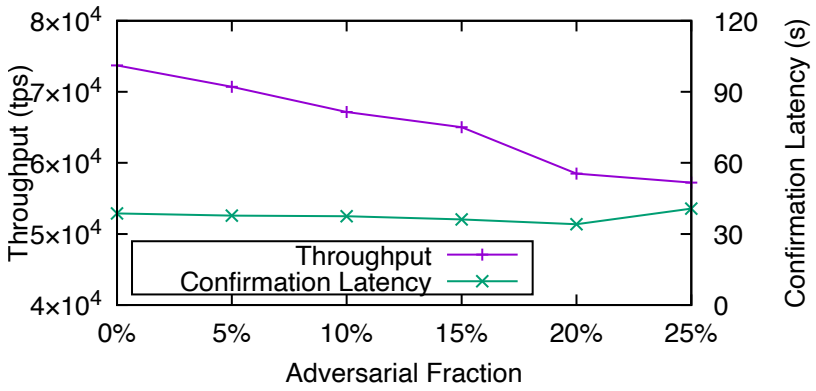


Figure 7-5: Performance of Prism under censorship attack.

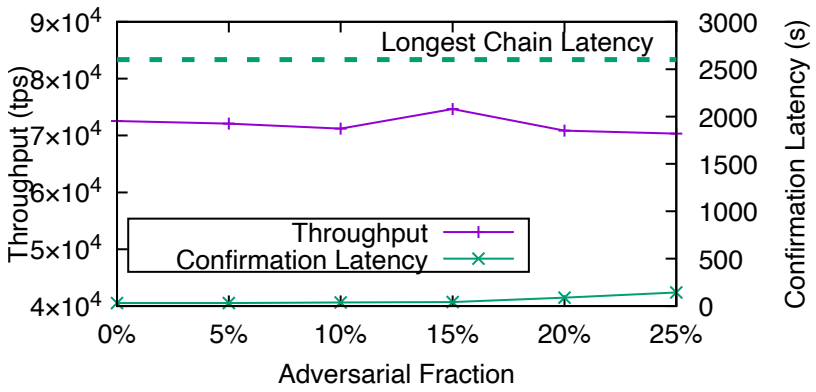


Figure 7-6: Performance of Prism under balancing attack. We also mark the confirmation latency of the longest chain protocol with the same security guarantee.

votes to be more evenly distributed among competing proposer blocks. In this case, clients need to wait for votes to grow deeper in order to confirm a proposer leader, resulting in longer confirmation latency. Figure 7-6 shows that the confirmation latency grows as the active adversarial fraction increases. But even when 25% clients are malicious, the confirmation latency is still more than 10× better than the longest chain protocol. Meanwhile, the throughput stays stable, because such attack only targets voter blocks.

Chapter 8

Conclusion

This thesis presented the implementation and evaluation of a Bitcoin-like system based on the Prism consensus protocol. Our implementation supports over 70,000 transactions per second at a confirmation latency of tens of seconds with Bitcoin-level security. Our results validate the theoretical analysis of the Prism protocol, and highlight the importance of optimizing transaction execution and the databases for high throughput. We also demonstrated experimentally that Prism is robust to several active attacks, and showed that a simple jittering approach is effective at mitigating spamming.

There are several avenues for future work. Our current implementation uses a UTXO-based scripting layer, and extending it to a more complex scripting layer for smart contracts is of interest. As described in §6.2, parallelizing transaction execution (via *scoreboarding*) was vital in achieving high throughput. The ability to parallelize transaction execution for smart contracts will be key to exploiting the high throughput provided by Prism consensus. Other extensions include methods to bootstrap new users and support light clients who only download the block headers (but not full blocks). Efficient bootstrapping is particularly important in a protocol like Prism that operates near network capacity, since expecting a new user to download and process all the old blocks is not practical.

Bibliography

[1] algorand/go-algorand: Algorand's official implementation in go. https://github.com/algorand/go-algorand.

[2] Global ping statistics. https://wondernetwork.com/pings/.

[3] ivicanikolicsg/ohie: Ohie - blockchain scaling. https://github.com/ivicanikolicsg/OHIE.

[4] Rocksdb | a persistent key-value store. https://rocksdb.org.

[5] Rust implementation of the prism consensus protocol. `https://github.com/yangl1996/prism-rust`.

[6] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Deconstructing the blockchain to approach physical limits. *accepted to ACM CCS 2019, arXiv:1810.08092*, 2018.

[7] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[8] Vitalik Buterin. Ethereum 2.0 mauve paper. In *Ethereum Developer Conference*, volume 2, 2016.

[9] Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. *arXiv preprint arXiv:1707.01873*, 2017.

[10] Lin William Cong and Zhiguo He. Blockchain disruption and smart contracts. *The Review of Financial Studies*, 32(5):1754–1797, 2019.

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

[12] Christian Decker, Rusty Russell, and Olaoluwa Osuntokun. eltoo: A simple layer2 protocol for bitcoin. *White paper: https://blockstream. com/eltoo. pdf*, 2018.

[13] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In Andrzej Pelc and Alexander A. Schwarzmann, editors, *Stabilization, Safety, and Security of Distributed Systems -*

17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings, volume 9212 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2015.

[14] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-ng: A scalable blockchain protocol. In *NSDI*, pages 45–59, 2016.

[15] Matthias Fitzi, Peter Gaži, Aggelos Kiayias, and Alexander Russell. Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition. Cryptology ePrint Archive, Report 1119, 2018.

[16] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 281–310. Springer, 2015.

[17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 51–68. ACM, 2017.

[18] GG Gueta, I Abraham, S Grossman, D Malkhi, B PINKAS, MK REITER, DA SEREDINSCHI, O TAMIR, and A TOMESCU. Sbft: a scalable decentralized trust infrastructure for blockchains, 2018, 1804.

[19] Garrick Hileman and Michel Rauchs. Global cryptocurrency benchmarking study. *Cambridge Centre for Alternative Finance*, 33, 2017.

[20] Erol Kazan, Chee-Wee Tan, and Eric TK Lim. Value creation in cryptocurrency networks: Towards a taxonomy of digital business models for bitcoin companies. In *PACIS*, page 34, 2015.

[21] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[22] Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1:11, 2014.

[23] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *International Conference on Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[24] Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:1805.03870*, 2018.

[25] Marta Lokhava, Giuliano Losa, David Maziéres, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th Symposium on Operating Systems Principles*. ACM, 2019.

[26] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.

[27] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42. ACM, 2016.

[28] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[29] Christopher Natoli and Vincent Gramoli. The balance attack against proof-of-work blockchains: The r3 testbed as an example. *arXiv preprint arXiv:1612.09426*, 2016.

[30] US NIST. Descriptions of sha-256, sha-384 and sha-512, 2001.

[31] Marc Pilkington. 11 blockchain technology: principles and applications. *Research handbook on digital transformations*, 225, 2016.

[32] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[33] Pandian Raju, Soujanya Ponnapalli, Evan Kaminsky, Gilad Oved, Zachary Keener, Vijay Chidambaram, and Ittai Abraham. mlsm: Making authenticated storage faster in ethereum. In *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[34] Y Sompolinsky, Y Lewenberg, and A Zohar. Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptology ePrint Archive*, 2016:1159.

[35] Y Sompolinsky and A Zohar. Phantom: A scalable blockdag protocol, 2018.

[36] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *International Conference on Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[37] James E. Thornton. 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*, AFIPS '64 (Fall, part II), pages 33–40, New York, NY, USA, 1965. ACM.

[38] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In Jay R. Lorch and Minlan Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019.*, pages 95–112. USENIX Association, 2019.

[39] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*, pages 45–54. IEEE, 2018.

[40] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[41] Haifeng Yu, Ivica Nikolic, Ruomu Hou, and Prateek Saxena. OHIE: blockchain scaling made simple. *CoRR*, abs/1811.12628, 2018.

Appendix A

Testbed Design

In this appendix, we provide the design details of our testbed that enables us to evaluate Prism on up to 1000 EC2 virtual machines. The testbed consists of a script written in 1000 lines of `Bash` that manages the EC2 cluster, and a tool written in 1200 lines of `Golang` that collects experimental data.

A.1 Working with an EC2 Cluster

The testbed runs on Amazon EC2's `c5d.4xlarge` instances. This type of instance has 16 vCPUs, 16 GB of RAM, 400 GB of NVMe SSD, and a 10 Gbps network interface. Before starting an experiment, our Bash script calls the API of AWS to start the required instances. We noticed that sometimes the AWS datacenter (US East, Ohio) may run out of capacity and was unable to provide the instances. The situation usually resolves within a few minutes as the datacenter auto-provisions more instances of the type, and our script is written to handle this issue.

After the instances are started, the script queries the IP addresses of the instances through AWS API and writes to the local SSH config file to enable pubkey-authenticated login. It then generates a payload for each instance, including the binary of the Prism client and the full command that starts the client. To deploy the payload, instead of sending to individual servers through `scp`, it first uploads the payload to AWS S3 and controls each VM to download the payload from S3, avoiding sending

the large binary files through the internet multiple times. It then mounts the NVMe storage on each VM, and configures the network bandwidth limiter and sets up an artificial delay to mimic the real internet. Specifically, we limit the total egress and ingress bandwidth respectively. While it is straightforward to shape the egress traffic by setting up `qdisc`, Linux does not allow traffic shaping of ingress traffic directly. As a solution, we forward all ingress traffic to an `ifb` device, and set up `qdisc` on this device. Finally, we tune the TCP send and receive buffer sizes to make sure the bandwidth is fully utilized.

In addition to provisioning EC2 VMs, the Bash script also has a few features to help us debug the testbed. Specifically, we found that being able to easily profile the program using Flamegraph is very useful for performance debugging and encourages us to fine-tune the program, especially since our local development machine alone does not have the hardware resource to achieve a high transaction throughput and forbids us to reproduce performance issues locally. Also, we added a function to remotely check the correctness of the system, e.g. whether all instances reach consensus and agree on the same UTXO set. This feature allowed us to capture many obscure race conditions.

A.2 Monitoring the performance

To monitor the experiment, we wrote a tool in Golang that communicates with the HTTP API of our Prism client. It periodically queries the clients and displays the following performance metrics: generated transactions, confirmed transactions, deconfirmed transactions, local network queue length, mined blocks, local block propagation delay, received blocks, confirmation latency, and forking. It also stores the time-series data in a local round-robin database and plots the data in real-time. We found that having the ability to monitor many metrics of the system is very useful for debugging, especially when the cluster involves hundreds of VMs. For example, we discovered a performance bug due to bad usage of Mutex when we noticed unusual spikes of network queuing latency. Also, collecting time-series data in real-time allows

us to display the results in a Grafana dashboard during presentations and demos.

Appendix B

Confirmation Rule

In this appendix we give the detailed calculation of the confidence intervals of the votes a proposer block receives. It is used when confirming a leader proposer block, as mentioned in §5.3.

Consider the scenario where there are n proposer blocks at level l, and let $\mathcal{P} = \{B_1^P, B_2^P, \ldots, B_n^P\}$ denote the set of proposer blocks at level l. Now we want to count the number of votes each block will get with confidence $1 - \epsilon$.

Suppose B_i^P gets v_i votes. Here a vote stands for a voter block which is on the longest chain of its voter tree and votes for B_i^P. Let $\mathcal{V}_i = \{B_{i_1}^V, B_{i_2}^V, \ldots, B_{i_{v_i}}^V\}$ denote the set of votes that B_i^P has. For every vote $B_{i_j}^V$, let d_{i_j} denote its depth, which is the number of blocks appended to voter block $B_{i_j}^V$ in the longest chain, plus one.

Now, for each vote $B_{i_j}^V$ with depth d_{i_j}, we want to calculate the probability P_{i_j} of it being permanent. To do so, we consider a potential private double-spend attack, assuming an adversarial party is trying to overturn the voting results to elect a different proposer block B_A^P as the leader block of level l. Note that B_A^P could either be a block in $\mathcal{P}$, i.e. publicly known, or a block the adversary has privately mined but not released. To elect B_A^P as the leader block of level l, the adversarial party would need to mine its own voter chains to overturn some existing votes to vote for B_A^P.

We want to compute the probability of this happening. However, we do not know when the adversary started mining voter blocks for B_A^P. Notice that the adversary has no incentive to mine voter blocks for B_A^P until B_i^P has been mined and released. Since

the honest nodes are always releasing blocks, we can use the average depth of the votes for B_i^P in the public voter trees to estimate the time passed since B_i^P was released, hence bounding the expected number of votes the adversary could have accumulated on their private fork in the same amount of time. That is, since block inter-arrivals are exponentially distributed, the number of blocks mined since block B_i^P was proposed is a Poisson random variable, with rate equal to its mean. This quantity can be related to the time elapsed since B_i^P was released via the block mining rate.

More precisely, as an honest node, we assume the fraction of adversarial hashing power is β, and we can empirically estimate the average depth of existing public votes as $\bar{d} = \sum_{ij} d_{i_j} / \sum_i v_i$ and the forking rate α [1] of public voter chains. Since there are many voter chains, these estimates converge quickly to their true means. Then, we calculate the estimated average depth of a *private* voter chain, denoted as $\bar{d_A}$, to be

$$\bar{d_A} = \frac{\beta \bar{d}}{(1-\alpha)(1-\beta)}.$$

Here the $1/(1-\alpha)$ term accounts for forking in public voter chains and assumes that the malicious private voter chains do not fork. The $\beta/(1-\beta)$ term accounts for the ratio of hashing power between the honest users $(1-\beta)$ and the malicious users (β). This expected depth $\bar{d_A}$ can be used as an estimate of the rate of the Poisson random variable of the number of blocks in the adversary's private chain.

Since each voter chain follows the longest-chain rule, the calculation for P_{i_j} is the same as in Bitcoin

$$P_{i_j} = F_{\text{Pois}}(d_{i_j}; \bar{d_A}) - \sum_{k=0}^{d_{i_j}} f_{\text{Pois}}(k; \bar{d_A}) \frac{\beta}{1-\beta}^{d_{i_j}+1-k}.$$

Here $F_{\text{Pois}}(x; \lambda)$ is the cumulative distribution function and $f_{\text{Pois}}(x; \lambda)$ is the probability mass function of Poisson distribution with rate parameter λ. In this expression, the first term is the probability that the adversary has mined fewer than $d_{i_j} + 1$ blocks, in which case it cannot currently overtake the main chain. The second term computes,

[1] The fraction of blocks not on the longest chain out of all blocks.

for each possible length of the adversary's chain, the probability that the adversary overtakes the public voter chain in the future by mining faster.

Given P_{i_j}, we can now calculate the confidence interval of votes on each proposer block. For proposer block B_i^P and each of its votes $B_{i_j}^V$, let $\tilde{V}_{i_j}$ be the random variable where

$$\tilde{V}_{i_j} = \begin{cases} 1, & \text{if vote } B_{i_j}^V \text{ is secure forever (permanent)} \\ 0, & \text{if vote } B_{i_j}^V \text{ will be overturned} \end{cases}.$$

With some abuse of notation, let v_i be the random variable equal to the number of secure votes of B_i^P. We have

$$v_i = \sum_j B_{i_j}^V.$$

Note that $\tilde{V}_{i_j} \sim \text{Bernoulli}(P_{i_j})$. Then the lower confidence bound of votes on B_i^P (denoted as $\lfloor v_i \rfloor$) can be obtained by calculating the ϵ-quantile of random variable v_i.

In real-world implementations, given the complexity of such computation, its closed-form approximation may be used. We can approximate v_i using a Gaussian distribution $\mathcal{N}(\mu_i, \sigma_i^2)$ where

$$\mu_i = \sum_j P_{i_j}.$$

$$\sigma_i^2 = \sum_j P_{i_j}(1 - P_{i_j}).$$

Using the closed-form approximation of the quantile function of normal distribution, we have

$$\lfloor v_i \rfloor \approx \mu_i - \sigma_i \sqrt{\ln \frac{1}{\epsilon^2} - \ln \ln \frac{1}{\epsilon^2} - \ln (2\pi)}.$$

Now, we consider the upper confidence bound of votes on B_i^P (denoted as $\lceil v_i \rceil$). Here, we want to defend against the worst case where for each B_i^P, only $\lfloor v_i \rfloor$ votes are retained, and the adversarial party controls the remaining votes (we let $\lceil v_A \rceil$ denote the number of such votes). Recall that each voter chain can only vote for each proposer level once. For a system with m voter chains, we have

$$\lceil v_A \rceil = m - \sum_i \lfloor v_i \rfloor .$$

The adversarial party will use those votes to vote for B_A^P. Since B_A^P could be any block in $\mathcal{P}$, we have

$$\lceil v_i \rceil = \lfloor v_i \rfloor + \lceil v_A \rceil .$$

B_A^P could also be a block which the adversarial party mines but has not released. In such case, the upper bound of votes on B_A^P is just $\lceil v_A \rceil$. Finally, to select the leader of level l, we search for the block $B_L^P \in \mathcal{P}$ satisfying $\lfloor v_L \rfloor > \lceil v_i \rceil$ for every $i \neq L$ and $\lfloor v_L \rfloor > \lceil v_A \rceil$. In words, we select the block whose lower bound of votes is higher than the upper bound of any other known or unknown proposer block in the same level.

Compared to the one proposed in the original Prism protocol (§4.5.1 of [6]), this new confirmation rule differs in how it calculates the confidence interval of the number of votes a proposer block ultimately receives given the current depth of each vote. The original protocol sets different thresholds for the vote depth and counts the votes that are deeper than the threshold. Then it deducts the number of votes that may be reversed by an attacker w.r.t. the chosen depth threshold from the previous count and gets the number of secured votes. It then picks the maximal number of secured votes given different depth thresholds as the lower bound for the proposer block. Here, any vote that is not deeper than the threshold are considered unsettled and hence does not contribute towards confirmation. In our new confirmation rule, however, every vote contributes towards confirmation regardless of its depth. Each vote B_{ij}^V is treated as a Bernoulli random variable with parameter P_{i_j} ($1 - P_{i_j}$ being the probability that the attacker reverses the vote) and we calculate the number of votes that the attacker can flip *at the same time* given the required confirmation error probability ϵ. As a result, even if P_{i_j} is very low for a particular vote, it is still counted towards the lower bound of settled votes.

This new confirmation rule provides three main benefits. First, it provides a lower confirmation latency than the original protocol because every vote contributes regardless of its depth, as explained above. Second, it allows flexible selection of the confirmation error probability ϵ. We can change ϵ easily by setting the confidence level to be $1 - \epsilon$ while calculating the confidence interval of v_i. As a comparison, the

original protocol defines ϵ only w.r.t. β, the number of voter chains m, and a finite execution horizon assuming the protocol only executes for a finite duration. Hence, the original protocol does not provide a way to select ϵ on each individual client. Third, the new confirmation rule is more practical to implement than the original one. This is because the original protocol requires multiple scans of the votes assuming different depth thresholds. Such operation can be hard to implement considering that the depth of votes continuouly changes. As a comparison, the new confirmation rule only scans the votes once, and allows approximations to speed up the calculation.

Appendix C

Parameters Used in the Evaluation

Here we present the parameters used in the experiment in §7.1 for Prism (Table C.1, Table C.2), Algorand (Table C.3), Bitcoin-NG (Table C.4), and the longest chain protocol (Table C.5).

Table C.1: Parameters of Prism.

Parameter	Value
Transaction Block Size	228 transactions
Voter Block Size	1000 votes
Proposer Block Size	7000 references
Voter Chains (m)	1000
Transaction Mining Rate	350 Blocks/s
Voter Mining Rate	Table C.2
Proposer Mining Rate	Table C.2

Table C.2: Mining rate f of proposer and voter blocks for different β in Prism. The unit is Blocks/s.

β	Mining Rate (f)
0.20	0.535
0.33	0.185
0.40	0.097
0.42	0.081
0.43	0.069
0.44	0.054

Table C.3: Parameters of Algorand. Block Size: number of transactions in a block. Assembly Time: maximum time spent on assembling a block (this limit was never hit in the experiment). λ: expected time to reach consensus on block hash. Λ: expected time to reach consensus on the actual block. Detailed definition in [17].

Block Size	Assembly Time (s)	λ (s)	Λ (s)
1287	0.5	0.6	1.6
4366	0.8	1.2	3.0
8733	1.6	1.9	6.5
13100	1.6	1.9	10.0
17294	1.6	2.0	13.0
21504	1.9	2.3	16.0
42334	3.5	3.9	38.0
64614	5.0	5.4	56.0
85513	7.0	7.4	73.0
85836	7.0	7.4	68.0
103004	8.4	8.8	84.0
116580	9.5	9.9	99.0
133766	11.0	11.4	110.0

Table C.4: Parameters of Bitcoin-NG.

Parameter	Value
Key Block Mining Rate	0.10 Block/s
Micro Block Interval	15000 μs
Block Size	500 transactions

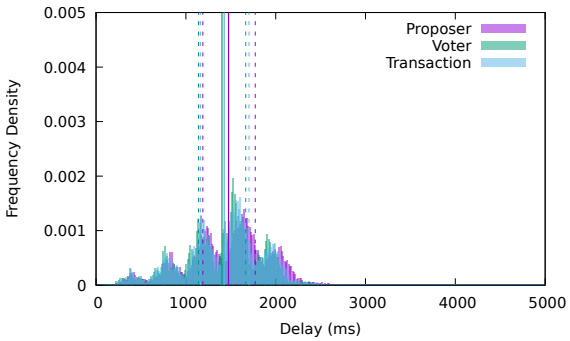
Table C.5: Mining rate f for different β and block sizes in the longest chain protocol. Here block sizes are in terms of transactions.

β	Block Size	Mining Rate (f)
0.20	10	0.404
	260	0.262
	1000	0.221
	4000	0.144
	10000	0.110
	20000	0.079
	60000	0.064
	200000	0.027
0.33	10	0.168
	260	0.117
	1000	0.119
	4000	0.065

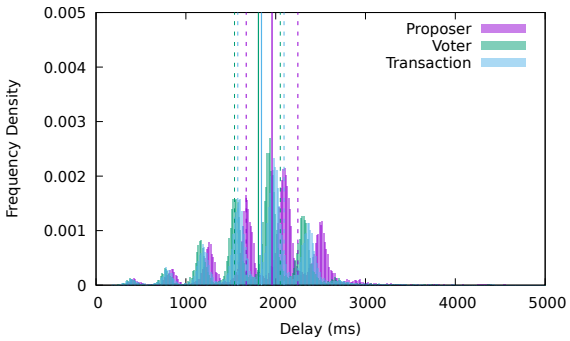
Appendix D

Block Propagation Delay Distribution

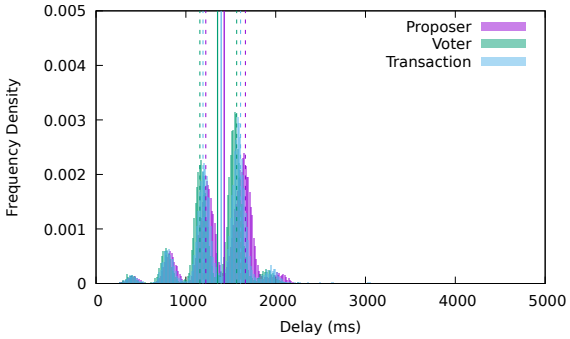
Here we present the distribution plots of the block propagation delay (Δ) in topologies tested in our scalability experiment (§7.2). The data are shown in Figures D-1a, D-1b, D-1c, D-1d, D-1e. In each plot, the concrete lines mark the mean of the propagation delay of that type of blocks, and the dashed lines mark the 25% and 75% quantiles. Comparing Figures D-1a, D-1c, D-1e we observe that as long as the network diameter is kept constant, the block propagation delay is barely affected by the increase of clients.



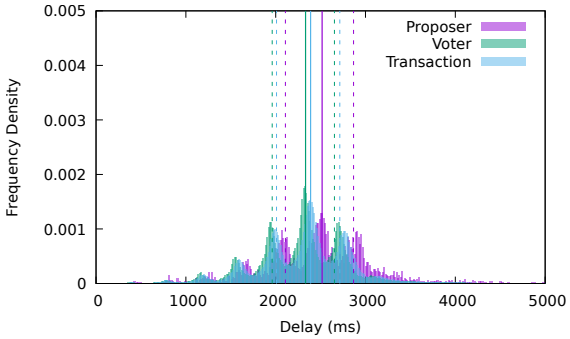
(a) Nodes= 100, Degree= 4, Diameter= 5



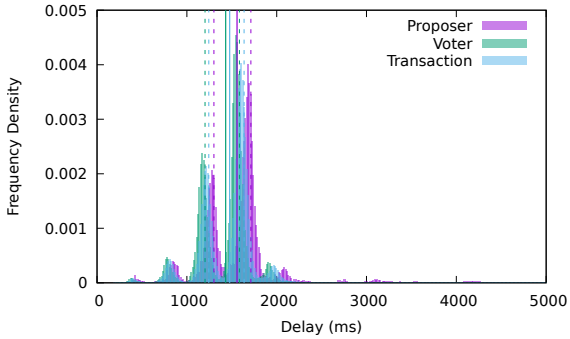
(b) Nodes= 300, Degree= 4, Diameter= 7



(c) Nodes= 300, Degree= 6, Diameter= 5



(d) Nodes= 1000, Degree= 4, Diameter= 9



(e) Nodes= 1000, Degree= 8, Diameter= 5

Figure D-1: Block propagation delay in the testbed.