A FULL-STACK STUDY OF BLOCKCHAINS ON SECURITY, PERFORMANCE, AND INCENTIVE

BY

GERUI WANG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Doctoral Committee:

Professor Pramod Viswanath, Chair
Professor Grigore Rosu
Assistant Professor Andrew Miller
Assistant Professor Ling Ren
Assistant Professor Kartik Nayak, Duke University

ABSTRACT

Blockchains have attracted colossal attention recently and are impacting various research areas while encountering several stiff challenges. This thesis makes contributions to blockchains by addressing three vital challenges in security, performance, and incentive.

First, the security challenge arises when malicious parties are beyond the threshold and launch a successful safety attack. A novel action called forensics is designed to detect the attackers with irrefutable proof. Moreover, the forensic support metric is mathematically formalized and systematically characterized for Byzantine fault-tolerant (BFT) protocols such as PBFT, HotStuff, and VABA, all of which possess strong forensic support. On the contrary, Algorand, a player replaceable protocol, is shown to have no forensic support. A novel protocol with both player replaceability and strong forensic support is presented to demonstrate the feasibility of having both properties in one BFT protocol.

Next, the performance challenge refers to the consensus layer bottleneck of permissionless blockchains. This bottleneck is removed by a novel consensus protocol called Prism, implemented in clients with a flexible interface to support smart contracts. Experiments show empirical evidence that Prism has removed the consensus layer performance bottleneck.

Finally, the incentive challenge is the phenomenon of token compounding in proof-of-stake (PoS) blockchains which leads to unfair reward distribution and discourages parties with a small number of tokens from joining PoS blockchains. A metric of fair reward distribution called equitability is mathematically formalized and analyzed for extant reward distribution mechanisms. Furthermore, a novel mechanism called the geometric reward is introduced and proved to be the most equitable.

ACKNOWLEDGMENTS

First and foremost I am extremely grateful to my advisor, Professor Pramod Viswanath for his patient support and invaluable advice during my Ph.D. study. His knowledge and insight have encouraged me in my academic research and daily life. I would also like to thank the thesis committee members, Professor Grigore Rosu, Professor Andrew Miller, Professor Ling Ren, and Professor Kartik Nayak for their support and advice on my thesis.

I would like to thank my collaborators, Professor David Tse, Professor Sreeram Kannan, Professor Mohammad Alizadeh, Professor Giulia Fanti, Professor Kartik Nayak, Professor Leonid Kogan, Professor Sewoong Oh, Professor Aggelos Kiayias, Dr. Vivek Bagaria, Dr. Matthias Fitzi, Dr. Nikos Leonardos, Peiyao Sheng, Lei Yang, Shuo Wang, Xuechao Wang, and Kathleen Ruan. It is my pleasure to work with them. I would also like to thank Ittai Abraham, Dahlia Malkhi, Zekun Li, David Wong, Avery Ching, Shaz Qadeer, Sam Blackshear, and Jovan Komatovic for their helpful discussions about the research topics of this thesis.

During my Ph.D. time, countless people have helped me, including, but not limited to, Weihao Gao, Jiaqi Mu, Hongyu Gong, Ranvir Rana, Hyeji Kim, Tarek Sakakini, Ashok Vardhan Makkuva, Pan Li, Zhuolun Xiang, Shiyu Liang, Songze Li, Siheng Pan, Ruiyang Chen, and Moitreya Chatterjee. I would like to express my thanks to them. I treasure the time when I studied on the University of Illinois campus and lived in Urbana and Champaign. I would like to thank the people there. In addition, I would like to thank the Illinois Computer Science department and the Graduate College for creating a nice and comfortable environment for studying and researching. I spent the last year of my Ph.D. program studying remotely due to the pandemic, and I would like to offer my appreciation to those who fight against the pandemic.

I would like to send my gratitude to Professor Andrew Chi-Chih Yao for creating the Yao class. My gratitude extends to the Yao class family and Tsinghua University where I did my undergraduate study and started academic research.

I would like to express my gratitude to my parents for their understanding and support in the past years. I could not finish my Ph.D. study without them.

Finally, I would like to convey my special appreciation to Yingyue for her love and support. Her elegance and passion always inspire me. Meeting her is the best thing that ever happened to me.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Background 1
1.2 Thesis Contributions 2
1.3 Related Work 4

CHAPTER 2 COMPOUNDING OF TOKEN IN PROOF-OF-STAKE BLOCKCHAINS 6
2.1 Introduction 6
2.2 Related Work 9
2.3 Models and Notation 10
2.4 Equitability under Honest Behavior 16
2.5 Strategic Behavior 22
2.6 Conclusion 27

CHAPTER 3 REMOVING CONSENSUS BOTTLENECK FOR PERMISSIONLESS BLOCKCHAIN SYSTEMS 29
3.1 Introduction 29
3.2 Related Work 30
3.3 Overview of Prism 32
3.4 Design and Implementation 35
3.5 Evaluation 40
3.6 Conclusion 48

CHAPTER 4 BLOCKCHAIN CONSENSUS PROTOCOL FORENSICS 50
4.1 Introduction 50
4.2 Related Work 53
4.3 Problem Statement and Model 55
4.4 Forensic Support for PBFT 57
4.5 Forensic Support for HotStuff 66
4.6 Forensic Support for VABA 79
4.7 Forensic Support for Algorand 83
4.8 Forensic Support for DiemBFT 87
4.9 Impossibility of Forensic Support for $n = 2t + 1$ 91
4.10 Conclusion 93

CHAPTER 5 PLAYER REPLACEABLE BLOCKCHAINS WITH FORENSIC SUPPORT 95
5.1 Introduction 95
5.2 Model and Definitions 98

5.3 Main Results: A Player Replaceable Protocol 100
5.4 An Extension: Reconfigurable Protocols 112
5.5 Related Work 116
5.6 Discussion 116

REFERENCES 118

CHAPTER 1: INTRODUCTION

1.1 BACKGROUND

Since Bitcoin's longest chain protocol [1] was invented in 2008, a family of distributed systems (systems that consist of multiple machines) known as *blockchain systems* has attracted interest from academia and industry. A blockchain is an abstraction of a chained sequence of data blocks containing payloads, while blockchain protocols refer to those distributed protocols that use the concept of blockchains, and blockchain systems refer to distributed systems that implement blockchain protocols. Most blockchain protocols achieve consensus, a crucial problem in distributed computing and distributed system research. In recent years, various new consensus protocols using or inspired by blockchains achieved the state of the art [2, 3, 4]. Meanwhile, blockchain systems have given rise to platforms of novel applications, such as cryptocurrency platforms (Bitcoin and Ethereum) and smart contract platforms (Ethereum, Hyperledger, and Diem).

The quintessence of distributed systems is *security*. In the lens of consensus, which is the core goal of most blockchain protocols, security means safety and liveness. Safety means any two non-faulty machines should agree on the same output, while liveness means client requests will eventually be output by non-faulty machines. Blockchain protocols are also fault-tolerant, that is, they continue to guarantee the security properties despite a certain number of machines fail. The best fault tolerance of state machine replication (the goal of many blockchains) is 1/3 or 1/2 of Byzantine (arbitrary) faulty machines under partial synchronous [5] or synchronous settings, respectively.

In practice, distributed systems also need to have good *performance*. The two significant metrics of performance are throughput (the number of processed transactions/requests per unit time) and latency (the time from the transaction gets to the proposal till the transaction is output). The throughput of Ethereum, the most popular permissionless smart contract platform currently, is around 20 tx/s (transaction per second).

Besides security and performance, blockchain systems have a distinct feature, namely decentralization. Decentralization refers to the absence of a primary party that controls the system. One of the intentions of decentralization is to encourage participation from diversified origins (e.g., different companies and organizations). To this end, the system should *incentivize* parties to join and stay in the system, otherwise they would stop participating or even leave the system, diminishing the degree of decentralization. For example, most blockchain systems have a type of digital resource called tokens, and the equitable sharing

of tokens among participants is an important incentive aspect.

Although blockchains show excellent promise of research and application, there are a few challenges in security, performance, and incentive.

Insecurity beyond fault tolerance threshold. Most existing works target 1/3 or 1/2 Byzantine fault tolerance (BFT) threshold, optimal under partial synchronous or synchronous settings. As a result, there is no security guarantee when the Byzantine fraction is beyond the threshold. For example, under partial synchrony, when 34% of machines are Byzantine faulty, they can produce a safety violation, i.e., disagreement among honest machines, leading to catastrophic outcomes. Furthermore, the faulty machines are not accountable for it, so there is no way to stop them after the fact.

Consensus bottleneck of throughput. Existing permissionless blockchain systems such as Bitcoin and Ethereum are based on the longest chain protocol, the original blockchain consensus protocol invented in 2008 [1]. It is well-known that the longest chain protocol suffers from poor throughput and latency performance. The throughput of Bitcoin and Ethereum is less than 10 tx/s and 20 tx/s, respectively, far less than the growing demand for cryptocurrency and smart contract applications. Hence, the performance of these platforms is limited by the consensus layer, and it is urgent to remove this performance bottleneck.

Inequitable token reward. For proof-of-stake (PoS) blockchain systems with tokens as an incentive, the token reward is distributed according to the stake that parties hold. A key issue, namely compounding, complicates this token reward mechanism. Compounding means that whenever a party earns a reward, it adds that reward to its stake, which increases its chances of reaping even more reward. Compounding leads to a rich-get-richer effect, causing dramatic concentration of tokens and disincentivizing parties with a small number of tokens.

1.2 THESIS CONTRIBUTIONS

This thesis addresses the challenges mentioned above with full-stack coverage. For the incentive challenge, Chapter 2 introduces the notion of equitability to mathematically study token compounding in PoS blockchain generally, not specific to any individual blockchain protocol or system. For the performance challenge, Chapter 3 implements permissionless blockchain systems with a new consensus protocol, namely Prism [6], to remove the consensus bottleneck. For the security challenge, Chapter 4 studies the algorithmic action of "forensics" when the number of faults is beyond the threshold, focusing on popular consensus protocols

(PBFT, HotStuff, and Algorand). Next, Chapter 5 studies the forensics topic concerning player replaceable blockchain consensus protocols. The highlights of these studies are as follows.

1.2.1 Compounding of Token in Proof-of-Stake Blockchains

Chapter 2 studies token compounding from the perspective of the block reward function. We define the equitability of a block reward function, which intuitively captures how much the fraction of tokens belonging to a party can grow or shrink (under that block reward function) compared to the initial fraction. We introduce a block reward function called the geometric reward function, which increases geometrically over time. We show that it is the most equitable PoS block reward function. We also study the effects of PoS pooling and strategic mining behaviors on token compounding.

1.2.2 Removing Consensus Bottleneck for Permissionless Blockchain Systems

With a focus on smart contract platforms, Chapter 3 presents the design and implementation of Prism that provides a flexible interface for connecting with two types of smart contract. We report experimental results from the implementation of two smart contract virtual machines, Ethereum VM (EVM) and MoveVM, on top of Prism. The results show that smart contract platforms built on Prism can perform without the consensus layer bottleneck.

1.2.3 Blockchain Consensus Protocol Forensics

Chapter 4 formalizes the forensics of blockchain consensus protocols by the definition of forensic protocol that aims to identify as many Byzantine faulty machines as possible and in an as distributed manner as possible, when the number of faults exceeds the security threshold and a security breach is mounted. *Forensic support* is defined as a property to measure the capability of the best forensic protocol for a consensus protocol. It is characterized for PBFT [7, 8], HotStuff [2], VABA [3] and Algorand [9], showing that there exist minor variants of each protocol for which the forensic supports vary widely. Strong forensic support capability is shown for DiemBFT [10] and an open-sourced forensic protocol implementation on Diem client [11] is provided. Finally, an impossibility result is shown for blockchain consensus protocols designed for the synchronous setting.

1.2.4 Player Replaceable Blockchain with Forensic Support

Forensics encounters an obstacle when it concerns player replaceable blockchain consensus protocols, where protocol participants are substituted randomly every step of the protocol: forensic protocol cannot identify malicious participants who perform incompatible actions because participants are replaced. Chapter 5 proposes a novel player replaceable blockchain consensus protocol to overcome this obstacle and to provide strong forensic support. Another class of protocols called reconfigurable protocols has a similar difficulty regarding forensics. These protocols change participants every epoch on rules that protocol designers can flexibly specify. The idea in the player replaceable protocol is adapted to reconfigurable protocols to enhance their forensic support.

1.3 RELATED WORK

1.3.1 Consensus Protocols

In consensus protocols, a set of machines (also known as nodes, participants, or replicas) reach an agreement on a value or a sequence of values with the presence of faulty machines. The problem of reaching an agreement on a single value is known as Agreement [12] and reaching on an ever-growing, linearly ordered log of values is known as State Machine Replication (SMR) [13]. Byzantine machines are faulty machines that deviate from the protocol arbitrarily [14], and blockchain consensus protocols focus on Byzantine fault-tolerant (BFT) consensus protocols. The seminal work of [15] and [5] shows that it is impossible to solve consensus with any crash faults deterministically under asynchrony or with one-third Byzantine faults out of all machines under partial synchrony. PBFT [7, 8] is the first practical BFT SMR protocol in the partial synchronous setting, with cubic communication complexity of view change (reduced to quadratic by some variants). Following the classical approaches, blockchain inspires new BFT protocols. HotStuff [2] is a partial synchronous SMR protocol that enjoys a linear communication of view change and optimistic responsiveness. Tendermint [16] is a partial synchronous SMR protocol with a linear view change that uses peer-to-peer gossiping. In the asynchronous setting, references [17, 18] are pioneers in solving Byzantine consensus by using randomization. Reference [19] defines the problem Validated Byzantine Agreement and gives a solution in asynchrony with asymptotically optimal round number and expected cubic word communication. A recent work [3] solves it with asymptotically optimal communication (quadratic) and round number. Synchronous protocols such as [4, 20] aim at optimal latency. Algorand [9, 21] designs a committee self-selection mecha-

nism, and the Byzantine Agreement protocol run by the committee decides the output for all replicas. These BFT Agreement and SMR protocols are referred as classical-style consensus protocols. And BFT protocols usually specifically refer to classical-style consensus protocols.

Another family of SMR protocol is born from the longest chain protocol [1]. Protocols such as Bitcoin-NG [22], GHOST [23], OHIE [24], and Conflux [25] are examples of this family. Also, there are proof-of-stake protocols such as Ouroboros [26], Ouroboros Praos [27] and Snow White [28]. These longest-chain-style consensus protocols differ from classical-style consensus in that the former is often permissionless by nature whereas the latter is often permissioned.

Many consensus protocols have a leader/proposer election mechanism that decides the responsibility of proposing values for consensus. The vanilla leader election mechanism for permissioned protocols is that each participant has the same chance of being elected, and the election is either deterministic or randomized. This mechanism is common for classical-style protocols such as PBFT and HotStuff. Another mechanism requires participants to solve a cryptographic puzzle and use the solution as a *proof* of the leader election result. When the protocol uses a computational puzzle whose solving probability is decided by computational power, the mechanism is called proof-of-work (PoW). The longest chain protocol, Bitcoin-NG, GHOST, OHIE, and Conflux are examples of PoW. When the protocol uses a puzzle whose solving probability is decided according to the participants' stake, the mechanism is called proof-of-stake (PoS). Algorand, Ouroboros, Ouroboros Praos, and Snow White are examples of PoS.

1.3.2 Practical Systems

In 2008, Satoshi Nakamoto invented Bitcoin [1], the first blockchain cryptocurrency system. Ethereum [29] is a cryptocurrency system that supports smart contracts, which run on Ethereum virtual machine (EVM). Both Bitcoin and Ethereum adopt the longest chain protocol [1], and are permissionless blockchain systems. Diem (formerly known as Libra) is a permissioned blockchain system that also supports smart contracts that run on MoveVM. Hyperledger Fabric [30] proposes the execute-order-validate paradigm that is different from standard blockchain systems, and it can choose a consensus protocol in the "order" step.

CHAPTER 2: COMPOUNDING OF TOKEN IN PROOF-OF-STAKE BLOCKCHAINS

Proof-of-stake (PoS) is a promising approach for designing efficient blockchains, where proposers are randomly chosen with probability proportional to their stakes. A primary concern in PoS systems is the "rich getting richer" effect, whereby wealthier nodes are more likely to get elected, and hence reap the block reward, making them even wealthier. In this chapter, we introduce the notion of equitability which quantifies how much a proposer can amplify its stake compared to its initial investment. Even with everyone following protocol (i.e., honest behavior), we show that existing methods of allocating block rewards lead to poor equitability, as does initializing systems with small stake pools and/or large rewards relative to the stake pool. We identify a *geometric* reward function, which we prove is maximally equitable over all choices of reward functions under honest behavior and bound the deviation for strategic actions. The proofs involve the study of optimization problems and stochastic dominances of Pólya urn processes.

We introduce this work in §2.1 and discuss related works in §2.2. In §2.3, we present our model. In §2.4, we study equitability under honest behavior. In §2.5, we study the effects of strategic behavior on equitability. We conclude this chapter in §2.6.

This chapter is a joint work with Giulia Fanti, Leonid Kogan, Sewoong Oh, Kathleen Ruan, and Pramod Viswanath published as reference [31].

2.1 INTRODUCTION

A central problem in blockchain systems is that of block proposal: how to choose which block should be appended to the global blockchain. Many blockchains use a proposal mechanism by which one node is randomly selected as leader (or *block proposer*). This leader gets to propose the next block in exchange for a token reward — typically a combination of transaction fees and a freshly-minted *block reward* which is chosen by the system designers. Early cryptocurrencies, including Bitcoin, mainly used a leader election mechanism called *proof of work* (PoW). Under PoW, all nodes execute a computational puzzle. The node who solves the puzzle first is elected leader. PoW is quite robust to security threats, but energy-inefficient, consuming more energy than developed nations [32].

An appealing alternative to PoW is called *proof-of-stake* (PoS). In PoS, proposers are not chosen according to their computational power, but according to the stake they hold in the cryptocurrency. For example, if Alice has 30% of the tokens, it is selected as the next proposer with probability 0.3. Although the idea of PoS is both natural and energy-efficient,

the research community is still grappling with how to design a PoS system that provides security while also incentivizing nodes to act as network validators. Part of incentivizing validators is simply providing enough reward (in expectation) to compensate their resource usage. However, it is also important to ensure that validators are treated fairly compared to their peers. In other words, they cannot only be compensated adequately on average. The variance also matters.

This observation is complicated in PoS systems by a key issue that does not arise in PoW systems: *compounding*. Compounding means that whenever a node (Alice) earns a proposal reward, that reward is added to its account, which increases its chances of being elected in the future and reaping even more rewards. This leads to a rich-get-richer effect, causing dramatic concentration of wealth (token).

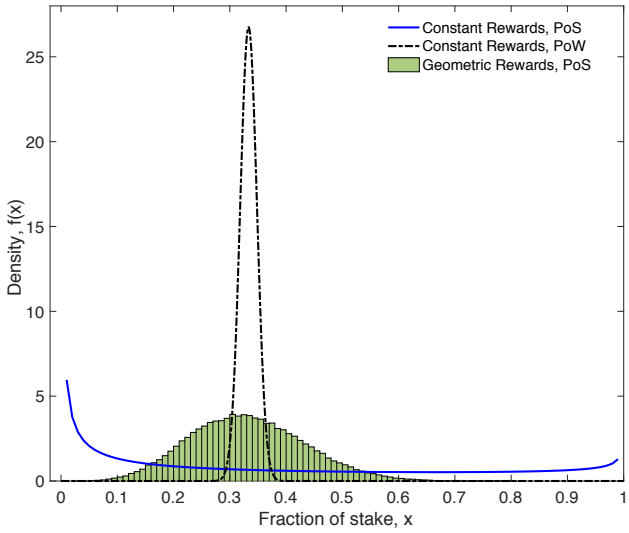


Figure 2.1: Fractional stake distribution of a party that starts with 1/3 of the stake in a system initialized with Bitcoin's financial parameters. Results of geometric reward PoS and constant reward PoW are shown after $T = 1,000$ blocks.

To see this, consider what would happen if Bitcoin were a PoS system. Bitcoin started with an initial stake pool of 50 BTC, and the block reward was fixed at 50 BTC/block for several years. Under these conditions, suppose a party A starts with $\frac{1}{3}$ of the stake. Using a basic PoS model described in §2.3, A's stake would evolve according to a standard Pólya urn process [33], converging almost surely to a random variable with distribution Beta($\frac{1}{3}, \frac{2}{3}$) [34], (blue solid line in Figure 2.1). In this example, compounding gives A a high probability of accumulating a stake fraction near 0 or 1. This is highly undesirable because the proposal incentive mechanism should not unduly amplify or shrink one party's fraction of stake. Notice that this is not caused by an adversarial or strategic behavior, but by the

randomness in the PoS protocol combined with compounding.

In PoW, on the other hand, the analogue would be for party A to hold $1/3$ of the computational power. In that case, A's stake after T blocks would be instead binomially distributed with mean $50\,T/3$ (black dashed line in Figure 2.1). Notice that the binomial (PoW) stake distribution concentrates around $1/3$ as $T \to \infty$, so if A contributes $1/3$ of stake at the beginning, it also reaps $1/3$ of the rewards in the long term.[1] Among randomized protocols that choose proposers independently at each time slot, the binomial distribution is the best we can hope for. It represents the setting where party A wins each block with probability equal to its initial stake. A natural question is whether we can achieve this PoW baseline distribution in a PoS system with compounding.

We study this question from the perspective of the block reward function. Most cryptocurrencies today use a *constant block reward* function like Bitcoin's, which remains fixed over a long timespan (e.g., years). We ask how a PoS system's choice of block reward function can affect concentration of wealth, and whether one can achieve the PoW baseline stake distribution simply by changing the block reward function. This chapter has five main contributions:

1. We define the *equitability* of a block reward function which intuitively captures how much the fraction of total stake belonging to a node can grow or shrink (under that block reward function), compared to the node's initial investment.
2. We introduce an alternative block reward function called the *geometric reward function* whose rewards increase geometrically over time. We show that it is the most equitable PoS block reward function by showing that it is the unique solution to an optimization problem on the second moment of a time-varying urn process. This optimization may be of independent interest. We note that despite optimizing equitability, geometric rewards do not achieve the PoW baseline stake distribution — this is the *inherent* price we pay for the efficiency of PoS compared to PoW. The green histogram in Figure 2.1 illustrates the empirical, simulated stake distribution when geometric rewards are used for $1\,000$ blocks, with total rewards as in the PoW example ($50 \times 1\,000$ units).
3. Borrowing ideas from mining pools in PoW systems, participants in a PoS system can form stake pools. We quantify the exact gains of stake pool formation in terms of equitability, which proves that participating in a stake pool can significantly reduce the compounding effect of a PoS system.
4. The effects of strategic behavior (e.g., selfish mining) on the rich-get-richer phe-

[1] Compounding can also happen in PoW if miners use their profits to purchase more mining equipment. However, this feedback loop is much slower and less direct than PoS compounding, so we approximate PoW by a system with no compounding.

nomenon is studied. We find that in general, compounding can exacerbate the efficacy of strategic behavior compared to PoW systems. However, these effects can be partially mitigated by carefully choosing the amount of block reward dispensed over some time period relative to the initial stake pool size.

5. Our analyses of the equitability of various reward functions provide guidelines for choosing system parameters — including the initial token pool size and the total rewards to dispense in a given time interval — to ensure equitability. We show that cryptocurrencies that start with large initial stake pools (relative to the block rewards being disseminated) can mitigate the concentration of wealth, both for constant and geometric reward schemes.

2.2 RELATED WORK

The compounding of wealth in PoS systems has been widely discussed in forum and blog posts [35, 36, 37], with recent work on *stake-bleeding attacks* exploiting exactly this property [38]. In this work, we quantify concentration of wealth through a new metric called equitability, which enables us to mathematically compare PoS to PoW, and different block reward schemes. As we discuss in §2.3, equitability is closely tied to the variance of a block reward scheme. Thus far, researchers and practitioners have reduced variance in block rewards through two main approaches: pooling resources (e.g., mining or stake pools) and proposing new protocols for disseminating block rewards.

Resource pooling is common in cryptocurrencies, e.g., in mining pools [39, 40]. In PoS systems, the analogous concept is stake pooling where nodes aggregate their stake under a single node and block rewards are shared across the pool. In §2.4.2, we show that the proposed geometric reward function is still the most equitable even if some parties are forming stake pools. Recent work [41] also studies stake pools and how to incentivize their formation through the design of reward mechanisms. Our work differs in that we aim to optimize equitability, whereas [41] aims to incentivize the formation of a target number of mining pools. Also, [41] does not consider the effects of compounding in PoS. A second variance reduction approach changes the block reward allocation protocol, and our work falls in this category. Two examples are Fruitchains [42], which spread block rewards evenly across a sequence of block proposers, and Ouroboros [26], which rewards nodes for being part of a block formation committee, even if they do not contribute to block proposal. Both of these approaches were proposed in order to provide incentive-compatibility for block proposers. They do not explicitly aim to reduce the variance of rewards, however, they implicitly reduce variance by spreading rewards across multiple nodes, thereby preventing

the randomized accumulation of wealth. In our work, instead of changing how block rewards are disseminated, we change the block reward function itself.

2.3 MODELS AND NOTATION

We provide a probabilistic model for the evolution of the stakes under a PoS system, and introduce a measure of fairness called *equitability*. We begin with a model of a chain-based proof-of-stake system with m parties: $\mathcal{A} = \{A_1, \ldots, A_m\}$. We assume that all parties keep all of their stake in the *proposal stake pool*, which is a pool of tokens that is used to choose the next proposer. We consider a discrete-time system, $n = 1, 2, \ldots, T$, where each time slot corresponds to the addition of one block to the blockchain. In reality, new blocks may not arrive at perfectly-synchronized time intervals, but we index the system by block arrivals. For any integer x, we use the notation $[x] := \{1, 2, \ldots, x\}$. For all $i \in [m]$, let $S_{A_i}(n)$ denote the total stake held by party A_i in the proposal stake pool at time n. We let $S(n) = \sum_{i=1}^{m} S_{A_i}(n)$ denote the total stake in the proposer stake pool at time n, and $v_{A_i}(n)$ denotes the *fractional stake* of node A_i at time n:

$$v_{A_i}(n) = \frac{S_{A_i}(n)}{S(n)}. \tag{2.1}$$

For simplicity, we normalize the initial stake pool size to $S(0) = 1$; this is without loss of generality as the random process is homogeneous in scaling both the rewards and the initial stake by a constant. Each party starts with $S_{A_i}(0) = v_{A_i}(0)$ fraction of the original stake. At each time $n \in [T]$, the system chooses a proposer node $W(n) \in \mathcal{A}$ so that

$$W(n) = \begin{cases} A_1 & w.p. \quad v_{A_1}(n) \\ \ldots & \\ A_m & w.p. \quad v_{A_m}(n). \end{cases} \tag{2.2}$$

Upon being selected as a proposer, $W(n)$ appends a *block*, or set of transactions, to the *blockchain*, which is a sequential list of blocks held by all nodes in the system. As compensation for this service, $W(n)$ receives a *block reward* of $r(n)$ stake, which is immediately added to its allocation in the proposer pool. I.e.,

$$S_{W(n)}(n+1) = S_{W(n)}(n) + r(n). \tag{2.3}$$

The reward $r(n)$ is freshly-minted, so it increases the total token pool size. We assume the total reward dispensed in time period T is fixed, such that $\sum_{n=1}^{T} r(n) = R$.

2.3.1 Modeling Assumptions

Our model implicitly makes several assumptions, such as a single proposer per time slot. Many cryptocurrencies have proposer election protocols that allow more than one proposer to be chosen per time slot (Bitcoin [1], PoSv3 [43], Snow White [44]). If two proposers are elected at time n, for example, then each can append its block to one block at height $n-1$; here the *height* of a block is its index in the blockchain. However, in these systems, only one leader can win the block reward since only one fork of the blockchain is ultimately adopted. Assuming the winner is chosen uniformly at random from the set of selected proposers, the dynamics of our Markov process remain unchanged.

Some cryptocurrencies (e.g., Qtum, Particl) choose proposer(s) as a function of the time slot *and* the preceding block. This does not affect our results in the honest setting (for the same reason as above), but it does increase the efficacy of strategic behavior like grinding [45] and selfish mining [40]. We discuss these implications in §2.5. Although we do not consider BFT style (classical-style) PoS protocols in this chapter, such protocols provide robustness to strategic behavior by forcing consensus on each block. They may also provide robustness to compounding, since block rewards can be shared among many nodes.

We have also assumed in this work that users instantly re-invest rewards into the proposer stake pool, for two reasons. (1) In PoS systems where users explicitly deposit stake, existing implementations automatically deposit rewards back into the stake pool. For example, the reference implementation of Casper the Friendly Finality Gadget (a PoS finalization mechanism proposed for Ethereum) automatically re-allocates all rewards back into the deposited stake pool [46]. (2) In other PoS systems, the stake pool is simply the set of all stake in the system, and is not separate from the pool of tokens used for transactions [43]. Hence as soon as a proposer earns a reward, that reward is used to calculate the next proposer (modulo some maturity period). The user is not actively re-investing block rewards — it just happens naturally. In practice, there may be a delay (maturity period) before the reward is counted; we do not model this effect.

2.3.2 Block reward choices

Many cryptocurrencies use Bitcoin's block reward schedule, which fixes the total supply of coins at about 21 million coins, and halves the reward every 210,000 blocks ($\approx$ 4 years) [47],

as illustrated by Figure 2.2. If we let T_i and R_i denote the ith block interval and the total reward, respectively, we can take $T_i = 210,000$ blocks, and $R_i = 50 \cdot \frac{1}{2^{i-1}} \cdot 210,000$. Several systems have adopted similar block rewards that are constant over long periods of time (e.g., Ethereum [48], ZCash [49], Dash [50], Particl [51]).

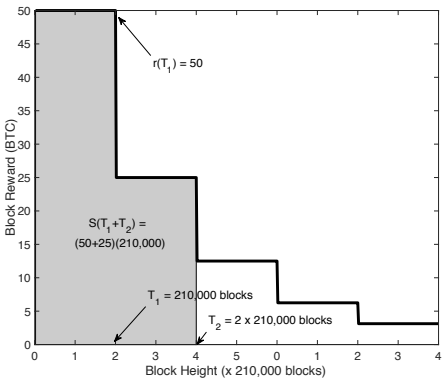


Figure 2.2: Bitcoin block rewards as a function of block height. The area of the shaded region gives the total stake after $T_1 + T_2$ time.

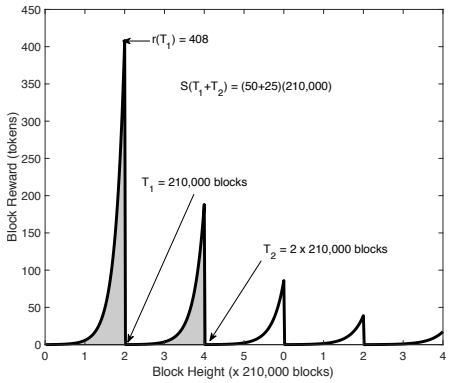


Figure 2.3: Geometric block rewards as a function of block height, using Bitcoin-based T_i and R_i values from Figure 2.2.

In this chapter, we revisit the question of how to choose $r(n)$. A key observation is that $r(n)$ must compensate nodes for the cost of proposing blocks. Many cryptocurrencies implicitly adopt the following maxim:

On short timescales, each block should yield the same block reward.

Notice that this maxim does not specify whether the value of a block reward is measured in tokens or in fiat. As illustrated earlier, most cryptocurrencies today measure value in tokens. We call this approach the *constant block reward*:

$$r_c(n) := \frac{R}{T}. \tag{2.4}$$

A natural alternative is to measure the block reward's value in fiat currency which depends on the cryptocurrency's valuation over time interval $[T]$. If we assume it to be constant, then the resulting reward function should give a constant fraction of the *total* stake at each time slot. We call this the *geometric* reward:

$$r_g(n) := (1+R)^{\frac{n}{T}} - (1+R)^{\frac{n-1}{T}}. \tag{2.5}$$

Figure 2.3 shows geometric block rewards as a function of time if we use the same T_i's and R_i's as in Figure 2.2, reflecting Bitcoin's block reward schedule.

2.3.3 Equitability

To compare reward functions, we define a metric called equitability. Consider the stochastic dynamic of the fractional stake of a party A that starts with $v_A(0)$ fraction of the initial total stake of $S(0) = 1$. We denote the fractional stake at time n by $v_{A,r}(n)$, to make the dependence on the reward function explicit. A straw-man metric for measuring fairness is the expected fractional stake at time T: i.e., if A contributes 10% of the proposal stake pool at the beginning of the time, then A should reap 10% of the total disseminated rewards on average. This metric is poor because PoS systems elect a proposer (in Equation (2.2)) with probability proportional to the fractional stake; this ensures that each party's expected fractional reward is equal to its initial stake fraction, for any block reward function. That is, $\forall n \in [T]$, $\mathbb{E}[v_{A,r}(n)] = v_A(0)$. This comes from the law of total expectation and the fact that

$$\begin{aligned}&\mathbb{E}[v_{A,r}(n) \,|\, v_{A,r}(n-1) = v] \\ &\quad = v\frac{v\,S(n-1) + r(n-1)}{S(n)} + \big(1-v\big)\frac{v\,S(n-1)}{S(n)} = v. \end{aligned} \tag{2.6}$$

Although all reward functions yield the same expected fractional stake, the choice of reward function can nonetheless dramatically change the distribution of the final stake, as seen in Figure 2.1. We therefore instead propose using the *variance* of the final fractional stake, $\mathrm{Var}(v_{A,r}(T))$, as an equitability metric. Intuitively, smaller variance implies less uncertainty and higher equitability:

Definition 2.1. For a positive vector $\boldsymbol{\varepsilon} \in \mathbb{R}^m$, we say a reward function $r : [T] \to \mathbb{R}^+$ over T time steps is $\boldsymbol{\varepsilon}$*-equitable* for $\boldsymbol{\varepsilon} = [\varepsilon_1, \ldots, \varepsilon_m]$ where $\varepsilon_i > 0$, if

$$\frac{\mathrm{Var}(v_{A_i,r}(T))}{v_{A_i}(0)(1 - v_{A_i}(0))} \;\leq\; \varepsilon_i \tag{2.7}$$

for all $i \in [m]$. For two reward functions $r_1 : [T] \to \mathbb{R}^+$ and $r_2 : [T] \to \mathbb{R}^+$ with the same total reward, $\sum_{n=1}^{T} r_1(n) = \sum_{n=1}^{T} r_2(n)$, we say r_1 is more *equitable* than r_2 for player $i \in [m]$ if

$$\mathrm{Var}\big(v_{A_i,r_1}(T)\big) \;\leq\; \mathrm{Var}\big(v_{A_i,r_2}(T)\big)\,, \tag{2.8}$$

when both random processes start with the same initial fraction of $v_{A_i}(0)$.

The normalization in Equation (2.7) ensures the left-hand side is at most one, as we show in Remark 2.1. It also cancels out the dependence on the initial fraction $v_A(0)$ such that the

left-hand side only depends on the reward function r and the time T, as shown in Lemma 2.1.

Remark 2.1. When starting with an initial fractional stake $v_A(0)$, the maximum achievable variance is

$$\sup_{T\in\mathbb{Z}^+}\sup_{r}\mathrm{Var}(v_{A,r}(T)) \;=\; v_A(0)(1-v_A(0))\;, \tag{2.9}$$

where the supremum is taken over all positive integers T and reward function $r:[T]\to\mathbb{R}^+$.

Proof. We first prove the converse, $\mathrm{Var}(v_{A,r}(T)) \le v_A(0)(1-v_A(0))$ for all T and r. This comes from the fact that $\mathbb{E}[v_{A,r}(T)] = v_A(0)$, and $v_{A,r}(T)$ is bounded below by zero and above by one. Maximum variance is achieved when all probability mass is concentrated on the boundary of zero and one.

We prove the achievability, by constructing a simple constant reward function with total reward $R = T^2$ increasing super-linearly in T. From the variance computation of a constant reward function in Equation (2.34), it follows that $\lim_{T\to\infty}\mathrm{Var}(v_{A,r_c}(T)) = v_A(0)(1-v_A(0))$. QED.

From the analysis of a time-dependent Pólya's urn model, we know the variance satisfies the following formula [52].

Lemma 2.1. Let $e^{\theta_n} \triangleq S(n)/S(n-1)$, then

$$\mathrm{Var}(v_{A,r}(T)) = \big(v_{A,r}(0) - v_{A,r}(0)^2\big)\Big(1 - \frac{S(0)^2}{S(T)^2}\prod_{n=1}^{T}(2e^{\theta_n}-1)\Big)\;. \tag{2.10}$$

Proof. Let $e^{\theta_n} \triangleq S(n)/S(n-1)$ and $r(n) = S(n+1) - S(n)$, then

$$\begin{aligned}
\mathbb{E}[v_{A,r}(n+1)^2|v_{A,r}(n)] &= v_{A,r}(n)\Big(\frac{S(n)v_{A,r}(n)+r(n)}{S(n+1)}\Big)^2 + (1-v_{A,r}(n))\Big(\frac{S(n)\,v_{A,r}(n)}{S(n+1)}\Big)^2 \\
&= \frac{(S(n)^2 + 2r(n)S(n))v_{A,r}(n)^2 + r(n)^2 v_{A,r}(n)}{S(n+1)^2} \\
&= (2e^{-\theta_{n+1}} - e^{-2\theta_{n+1}})v_{A,r}(n)^2 + (e^{-\theta_{n+1}}-1)^2 v_{A,r}(n)\;.
\end{aligned} \tag{2.11}$$

It follows that

$$\begin{aligned}\mathbb{E}[v_{A,r}(T)^2] - \mathbb{E}[v_{A,r}(T)] &= (2e^{-\theta_T} - e^{-2\theta_T})\big(\mathbb{E}[v_{A,r}(T-1)^2] - \mathbb{E}[v_{A,r}(T-1)]\big) \\ &= \big(\mathbb{E}[v_{A,r}(0)^2] - \mathbb{E}[v_{A,r}(0)]\big) \prod_{n=1}^{T} (2e^{-\theta_n} - e^{-2\theta_n}) \ . \end{aligned} \tag{2.12}$$

Hence,

$$\begin{aligned}\mathrm{Var}(v_{A,r}(T)) &= \mathbb{E}[v_{A,r}(T)^2] - \mathbb{E}[v_{A,r}(T)]^2 \\ &= \mathbb{E}[v_{A,r}(T)^2] - \mathbb{E}[v_{A,r}(T)] + \mathbb{E}[v_{A,r}(T)] - \mathbb{E}[v_{A,r}(T)]^2 \\ &= \mathbb{E}[v_{A,r}(T)^2] - \mathbb{E}[v_{A,r}(T)] + \mathbb{E}[v_{A,r}(0)] - \mathbb{E}[v_{A,r}(0)]^2 \\ &= \big(\mathbb{E}[v_{A,r}(0)] - \mathbb{E}[v_{A,r}(0)^2]\big)\Big(1 - \prod_{n=1}^{T}(2e^{-\theta_n} - e^{-2\theta_n})\Big) \\ &= \big(v_{A,r}(0) - v_{A,r}(0)^2\big)\Big(1 - \prod_{n=1}^{T} e^{-2\theta_n} \prod_{n=1}^{T}(2e^{\theta_n} - 1)\Big) \\ &= \big(v_{A,r}(0) - v_{A,r}(0)^2\big)\Big(1 - \frac{S(0)^2}{S(T)^2} \prod_{n=1}^{T}(2e^{\theta_n} - 1)\Big) \ . \end{aligned} \tag{2.13}$$

QED.

Although Definition 2.1 applies to an arbitrary number of parties, Lemma 2.1 implies that it is sufficient to consider a single party's stake. More precisely:

Remark 2.2. If reward function $r : [T] \to \mathbb{R}^+$ over T time steps is $\boldsymbol{\varepsilon}$-*equitable* for vector $\boldsymbol{\varepsilon} = [\varepsilon_1, \ldots, \varepsilon_m]$ where $\varepsilon_i > 0$, then r is also $\tilde{\boldsymbol{\varepsilon}}$-equitable, where

$$\tilde{\boldsymbol{\varepsilon}} \triangleq \mathbf{1} \cdot \min_{i \in [m]} \varepsilon_i, \tag{2.14}$$

with $\mathbf{1}$ denoting the vector of all ones.

As such, the remainder of this chapter will study equitability from the perspective of a single (arbitrary) party A. We will also describe reward functions as ε-equitable as shorthand for $\boldsymbol{\varepsilon}$-equitable, where $\boldsymbol{\varepsilon} = \mathbf{1} \cdot \varepsilon$. Note that even if the total reward R is fixed, equitability can differ dramatically across reward functions. In the example of Figure 2.1, the constant reward function is 0.5-equitable. Compared to the value 0.5, the geometric rewards of (2.5) have a smaller chance of losing all its fractional stake (i.e. $v_{A,r_g}(T) \approx 0$) or taking over the whole stake (i.e. $v_{A,r_g}(T) \approx 1$). It is 0.05-equitable in this example.

2.4 EQUITABILITY UNDER HONEST BEHAVIOR

In this section, we analyze the equitability of different block reward functions, assuming that every party is honest and the PoS system is *closed*, so no stake is removed or added to the proposal stake pool over a fixed time period T. Each party's stake changes only because of the block rewards it earns and compounding effects. We discuss the effects of strategic behavior in §2.5.

The metric of equitability leads to a core optimization problem for PoS system designers: given a fixed total reward R to be dispensed, how do we distribute it over the time T to achieve the highest equitability? Perhaps surprisingly, we show that this optimization has a simple, closed-form solution.

Theorem 2.1. For all $R \in \mathbb{R}^+$ and $T \in \mathbb{Z}^+$, the geometric reward r_g defined in (2.5) is the most equitable among functions that dispense R tokens over time T, jointly over all parties A_i, for $i \in [m]$.

Intuitively, geometric rewards optimize equitability because they dispense small rewards in the beginning when the stake pool is small, so a single block reward cannot substantially change the stake distribution. The rewards subsequently grow proportionally to the size of the total stake pool, so the effect of a single block remains bounded throughout the time period. We emphasize that the geometric reward function does not depend on the initial stake of the party A, and hence is universally most equitable for all parties in the system simultaneously.

Proof. Lemma 2.1 and Remark 2.2 imply that in order to show joint optimality over all parties, it is sufficient to show that for an arbitrary party A,

$$\mathrm{Var}\big(v_{A,r_g}(T)\big) \;\leq\; \mathrm{Var}\big(v_{A,r}(T)\big)\;, \tag{2.15}$$

for all $r \in \mathbb{R}^T$ such that $\sum_{n=1}^{T} r(n) = R$ and $r(n) \geq 0$ for all $n \in [T]$. To this end, we prove that r_g is a unique optimal solution to the following optimization problem:

$$\begin{aligned} \text{minimize}_{r\in\mathbb{R}^T} \quad & \mathrm{Var}(v_{A,r}(T)) && (2.16)\\ \text{s.t.} \quad & \sum_{n\in[T]} r(n) = R\;, \\ & r(n) \geq 0\;,\ \forall n \in [T]. && (2.17) \end{aligned}$$

Using Lemma 2.1, we have an explicit expression for $\mathrm{Var}(v_{A,r}(T))$. After some affine transformation and taking the logarithmic function of the objective, we get an equivalent opti-

mization of

$$\begin{aligned} \text{maximize}_{\theta \in \mathbb{R}^T} \quad & \sum_{n=1}^{T} \log(2e^{\theta_n} - 1) && (2.18) \\ \text{s.t.} \quad & \sum_{n \in [T]} \theta_n = \log(1+R)\ , \\ & \theta_n \geq 0, \forall n \in [T]. && (2.19) \end{aligned}$$

This is a concave maximization on a (rescaled) simplex. Writing out the KKT conditions with KKT multipliers λ and $\{\lambda_n\}_{n=1}^T$, we get $\forall n \in [T]$:

$$\begin{aligned} \frac{2e^{\theta_n}}{2e^{\theta_n} - 1} - \lambda_n - \lambda &= 0 && (2.20) \\ \lambda_n &\geq 0 && (2.21) \\ \theta_n \lambda_n &= 0 && (2.22) \end{aligned}$$

Among these solutions, we show that $\theta^* = ((\log(1+R))/T)\,\mathbb{1}$ is the unique optimal solution, where $\mathbb{1}$ is a vector of all ones. Consider a solution of the KKT conditions that is not θ^*. Then, we can strictly improve the objective by the following operation. Let $i, j \in [T]$ denote two coordinates such that $\theta_i = 0$ and $\theta_j \neq 0$. Then, we can create $\tilde{\theta}$ by mixing θ_i and θ_j, such that $\tilde{\theta}_n = \theta_n$ for all $n \neq i, j$ and $\tilde{\theta}_i = \tilde{\theta}_j = (1/2)\theta_j$. We claim that $\tilde{\theta}$ achieves a smaller objective function as $\log(2e^{\theta_j} - 1) < 2\log(2e^{\theta_j/2} - 1)$. This follows from Jensen's inequality and strict concavity of the objective function. Hence, θ^* is the only fixed point of the KKT conditions that cannot be improved upon.

In terms of the reward function, this translates into $S(n)/S(n-1) = (1+R)^{1/T}$ and $r(n) = (1+R)^{n/T} - (1+R)^{(n-1)/T}$.

QED.

2.4.1 Composition

The geometric reward function does not only optimize equitability for a single time interval. Consider a sequence $(T_1, R_1), \ldots, (T_k, R_k)$ of checkpoints, where T_i is increasing in i, and R_i denotes the amount of reward to be disbursed between time $T_{i-1} + 1$ and T_i (inclusive). These checkpoints could represent target inflation rates on a monthly or yearly basis, for instance. A natural question is how to choose a block reward function that optimizes equitability over all the checkpoints jointly. The solution is to iteratively and independently apply geometric rewards over each time interval, giving a block reward function like the one shown in Figure 2.3.

Theorem 2.2. Consider a sequence of checkpoints $\{(T_i, R_i)\}_{i\in[k]}$. Let $\tilde{R}_j := \sum_{i=1}^{j} R_i$. The most equitable reward function is

$$r(n) = (1+\tilde{R}_{i-1})\left(\left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}}\right)^{\frac{n-T_{i-1}}{T_i-T_{i-1}}} - \left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}}\right)^{\frac{n-1-T_{i-1}}{T_i-T_{i-1}}}\right) \tag{2.23}$$

for $n \in [T_{i-1}+1, T_i]$.

When there is only one checkpoint, Theorem 2.2 simplifies to Theorem 2.1. This implies that checkpoints can be chosen *adaptively*, i.e., they do not need to be fixed upfront to optimize equitability. In practice, the abrupt change in geometric block rewards at a checkpoint (Figure 2.3) may lead to miner/validator attrition [53]. Liquidity limits may slow down this attrition, but cannot stop it [44]. One option is that we can choose the block reward function not only based on equitability, but also concerning smoothness and/or monotonicity constraints. Another is that PoS blockchains could use geometric rewards only for the first epoch (when compounding poses the greatest risk), and then switch to a smoother block reward schedule of their choosing. We leave such exploration to future work.

Proof. By the same logic as the proof of Theorem 2.1, the optimization problem of interest can be written as

$$\begin{aligned} \text{maximize}_{\theta\in\mathbb{R}^{T_k}} \quad & \sum_{n=1}^{T_k} \log(2e^{\theta_n}-1) && (2.24)\\ \text{s.t.} \quad & \sum_{n=T_{i-1}+1}^{T_i} \theta_n = \log\left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}}\right) \ , \forall i \in [k] \ , \\ & \theta_n \geq 0, \forall n \in [T_k] \ , && (2.25) \end{aligned}$$

where recall that $\theta_n = \frac{S(n)}{S(n-1)}$, and we define $T_0 := 0$. Notice that this optimization problem is separable over the variables in different time intervals, so we can separately solve k optimization problems, each of the form

$$\begin{aligned} \text{maximize}_{\theta\in\mathbb{R}^{T_i-T_{i-1}}} \quad & \sum_{n=T_{i-1}+1}^{T_i} \log(2e^{\theta_n}-1) && (2.26)\\ \text{s.t.} \quad & \sum_{n=T_{i-1}+1}^{T_i} \theta_n = \log\left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}}\right) \ , \\ & \theta_n \geq 0, \forall n \in [T_{i-1}+1, T_i] \ , && (2.27) \end{aligned}$$

for each $i \in [k]$. Using the same KKT conditions as in Theorem 2.1, we get that $\theta_n^* = \frac{1}{T_i - T_{i-1}} \log(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}})$, which in turn implies that for $n \in [T_{i-1}+1, T_i]$,

$$S(n) = (1+\tilde{R}_{i-1}) \left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}} \right)^{(n-T_{i-1})/(T_i - T_{i-1})} \tag{2.28}$$

and

$$r(n) = (1+\tilde{R}_{i-1}) \left(\left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}} \right)^{\frac{n-T_{i-1}}{T_i - T_{i-1}}} - \left(\frac{1+\tilde{R}_i}{1+\tilde{R}_{i-1}} \right)^{\frac{n-1-T_{i-1}}{T_i - T_{i-1}}} \right). \tag{2.29}$$

QED.

2.4.2 Stake Pools

Participants also have the freedom to form stake pools, as explored in [39, 40, 41]. We show that stake pools reduce the variances of the fractional stake of all pool members, and quantify this gain. Consider a single party that owns $v_A(0)$ fraction of the stake at time $t = 0$. We know from Lemma 2.1 that the variance at time T is $\mathrm{Var}(v_{A,r}(T)) = \big(v_A(0) - v_A(0)^2\big)\Big(1 - \frac{S(0)^2}{S(T)^2} \prod_{n=1}^{T} (2e^{\theta_n} - 1)\Big)$. Consider a case where the same party now participates in a stake pool, where the pool P has $v_P(0)$ of the initial stake (including the contribution from party A), and every time the stake pool is awarded a reward for block proposal, the reward is evenly shared among the participants of the pool according to their stakes. The stake of party A under this pooling is denoted by $v_{\tilde{A}}(T)$, and it follows from Lemma 2.1 immediately that

$$\begin{aligned} \mathrm{Var}(v_{\tilde{A},r}(T)) &= \Big(\frac{v_A(0)}{v_P(0)}\Big)^2 \big(v_P(0) - v_P(0)^2\big)\Big(1 - \frac{S(0)^2}{S(T)^2} \prod_{n=1}^{T} (2e^{\theta_n} - 1)\Big) \\ &= \frac{1 - v_P(0)}{v_P(0)} \frac{v_A(0)}{1 - v_A(0)} \mathrm{Var}(v_{A,r}(T)) \,. \end{aligned} \tag{2.30}$$

Thus party A's variance reduces by a factor of $(v_P(0)/v_A(0))((1-v_A(0))/(1-v_P(0)))$ by joining a stake pool of size $v_P(0)$. Note that the variance is monotonically decreasing under stake pooling. In practice, stake pools can organically form as long as this gain in equitability exceeds the cost of pool formation. Applying Definition 2.1 to a single party A, an ε-equitable party A will achieve $\varepsilon \frac{v_A(0)(1-v_P(0))}{v_P(0)(1-v_A(0))}$-equitability by forming a stake pool. Further, geometric rewards are still the most equitable reward function in the presence of stake pools. This follows from the fact that the effect of pooling is isolated from the effect of the choice of the

reward function in Equation (2.30).

2.4.3 Practical parameter selection

The equitability of a system is determined by four factors: the number of block proposals T, choice of reward function r, initial stake of a party $v_A(0)$, and the total reward R. We have shown that geometric rewards optimize equitability. In this section, we study its dependence on T, $S(0)$, and R. Recall that without loss of generality, we normalized the initial stake $S(0)$ to be one. For general choices of $S(0)$, the total reward R should be rescaled by $1/S(0)$. The evolution of the fractional stakes is exactly the same for one system with $S(0) = 2$ and $R = 200$ and another with $S(0) = 1$ and $R = 100$. We assume here that the system designer can choose the total reward R, either by setting the initial stake size $S(0)$ and/or the total reward during T. We study how equitability trades off with the total reward R for different choices of the reward function.

Geometric rewards. For $r_g(n)$, we have $e^{\theta_n} = (1+R)^{1/T}$. It follows from Lemma 2.1 that

$$\frac{\mathrm{Var}(v_{A,r_g}(T))}{v_A(0) - v_A(0)^2} \;=\; 1 - \frac{(2(1+R)^{1/T} - 1)^T}{(1+R)^2}\,, \tag{2.31}$$

When R is fixed and T is increasing, we can distribute small amounts of rewards across T and achieve vanishing variance. On the other hand, if R increases much faster than T, then we are giving out increasing amounts of rewards per time slot and the uncertainty grows. This result follows from the above variance formula and is made precise in the following remark.

Remark 2.3. For a closed PoS system with a total reward $R(T)$ chosen as a function of T and a geometric reward function $r_g(n) = (1+R(T))^{n/T} - (1+R(T))^{(n-1)/T}$, it is sufficient and necessary to set $R(T)$ by using the following formula,

$$R(T) \;=\; \left(\left(\frac{1}{1 - \sqrt{\frac{\log(1/(1-\varepsilon))}{T}}} \right)^T - 1 \right) \big(1 + o(1)\big)\,, \tag{2.32}$$

in order to ensure ε-equitability asymptotically, i.e. $\lim_{T\to\infty} \frac{Var(v_{A,r_g}(T))}{v_A(0)(1-v_A(0))} = \varepsilon$.

Remark 2.3 follows from substituting the choice of $R(T)$ in the variance in Equation (2.31),

which gives

$$\begin{aligned}\lim_{T\to\infty} \frac{\mathrm{Var}(v_{A,r_g}(T))}{v_A(0)-v_A(0)^2} &= \lim_{T\to\infty} 1-\Big(1-\frac{\log(1/(1-\varepsilon))}{T}\Big)^T(1+o(1)).\\ &= \varepsilon\,, \end{aligned} \tag{2.33}$$

The limiting variance is monotonically non-decreasing in R and non-increasing in T, as expected from our intuition. For example, if R is fixed, one can have the initial stake $S(0)$ as small as $\exp(-\sqrt{T}/(\log T))$ and still achieve a vanishing variance. As the geometric reward function achieves the smallest variance (Theorem 2.1), the above $R(T)$ is the largest reward that can be dispensed while achieving a desired normalized variance of ε in time T (with initial stake of one). This scales as $R(T) \simeq (1+1/\sqrt{T})^T \simeq e^{\sqrt{T}}$. We need more initial stake or less total reward, if we choose to use other reward functions.

Constant rewards. In comparison, consider the constant reward function of Equation (2.4). As $e^{\theta_n} = (1+nR/T)/(1+(n-1)R/T)$, it follows from Lemma 2.1 that

$$\begin{aligned}\frac{\mathrm{Var}(v_{A,r_c}(T))}{v_A(0)-v_A(0)^2} &= 1-\frac{1+R+\frac{R}{T}}{1+R+\frac{R}{T}+\frac{R^2}{T}}\\ &= \frac{R^2}{(T+R)(1+R)}\,. \end{aligned} \tag{2.34}$$

Again, this is monotonically non-decreasing in R and non-increasing in T, as expected. The following condition immediately follows from Equation (2.34).

Remark 2.4. For a closed PoS system with a total reward $R(T)$ chosen as a function of T and a constant reward function $r_c(n) = R(T)/T$, it is sufficient and necessary to set

$$R(T) = \frac{\varepsilon\, T}{1-\varepsilon}\,(1+o(1))\,, \tag{2.35}$$

in order to ensure ε-equitability asymptotically as T grows.

By choosing a constant reward function, the cost we pay is in the size of the total reward, which can now only increase as $O(T)$. Compared to $R(T) \simeq e^{\sqrt{T}}$ of the geometric reward, there is a significant gap. Similarly, in terms of how small initial stake can be with fixed total reward R, constant reward requires at least $S(0) \simeq R/T$.

Comparison of Rewards. For $S(0) = 1$ and $R = 10$, Figure 2.4 illustrates the normalized variance of the two reward functions as a function of T, the total number of blocks. As

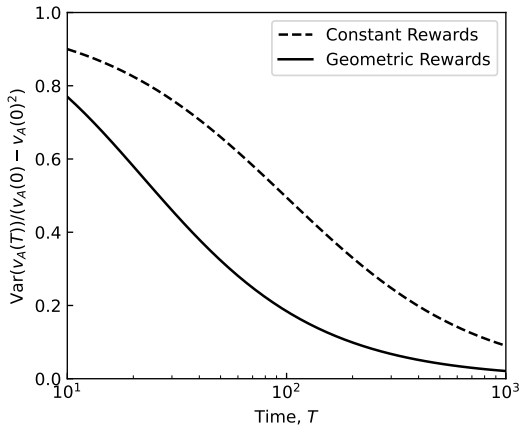


Figure 2.4: Normalized variance after dispensing $R = 10$ tokens over T blocks, under different reward schemes.

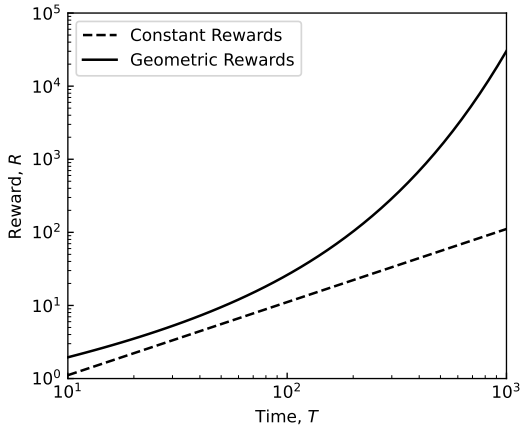


Figure 2.5: Amount of reward that can be dispensed over T blocks while guaranteeing a normalized variance of at most $\varepsilon = 0.1$.

expected, variance decays with T and geometric rewards exhibit lower normalized variance. Similarly, for a fixed desired (normalized) variance level of $\varepsilon = 0.1$, Figure 2.5 shows how the total reward grows as a function of time T. Notice that under constant rewards, the reward allocation grows linearly in T, whereas geometric rewards grow sub-exponentially while still satisfying the same equitability constraint. These observations add nuance to the ongoing conversation about how to initialize PoS cryptocurrencies. A recent lawsuit against Ripple highlighted that the large initial stake pool could put disproportionate power in the hands of the system designers [54]. While Ripple itself is not PoS, our results suggest that in standard PoS systems, a large initial stake pool can actually help to ensure equitability.

2.5 STRATEGIC BEHAVIOR

In practice, proposers can behave strategically to maximize their rewards (e.g., *selfish mining* [40, 55, 56]). In selfish mining, miners (proposers) do not immediately publish blocks, but build a private *withheld fork* of blocks. By eventually releasing a private chain that is longer than the main chain, the adversary can invalidate honest blocks. It gives the adversary a greater fraction of main chain blocks and wastes honest parties' effort. In this section, we show that such strategic attacks are exacerbated by the compounding effects of PoS, and geometric rewards do not provide adequate protection.

Modeling the space of strategic behaviors in PoS requires more nuance than the corresponding problem in PoW [55]. We restrict ourselves in this section to two parties: A which is adversarial, and H which is honest. Note that this restriction is without loss of generality, as H represents the collective set of multiple honest parties as their behavior is independent

of how many parties are involved in H. The adversarial party A can also represent the collective set of multiple adversarial parties, as having a single adversary A is the worst case when all adversaries are colluding. Throughout this section, we use the terms *adversarial* and *strategic* interchangeably.

Since A does not always publish its blocks on schedule, we distinguish the notion of a block slot (indexed by $n \in [T]$) and wall-clock time (indexed by $t \in [T]$). It will still be the case that each *block slot* n has a single leader $W(n)$ — in practice, this is determined by a distributed protocol — and a new block slot leader is elected at every tick of the wall clock (i.e., at a given time t, $W(n)$ is only defined for $n \leq t$). However, due to strategic behavior (i.e., the adversary can withhold its own blocks and override honest ones), it can happen that no block occupies slot n, even at time $t \geq n$; moreover, the occupancy of block slot n can change over time. Thus, unlike our previous setting, if we wait T time slots, the resulting chain may have fewer than T blocks. This is consistent with the adversarial model considered in PoS systems (e.g., Ouroboros [26]) which elect a single leader per block slot. Other PoS systems, like PoSv3 [43], choose an independent leader to succeed each block; such a PoS model can lead to even worse attacks, which we do not consider in this work.

The honest party and the adversary have two different views of the blockchain, illustrated in Figure 2.6. Both honest and adversarial parties see the *main chain* B_t; we let $B_t(n)$ denote the block (i.e., leader) of the nth slot, as perceived by the honest nodes at time t. If a block slot n does not have an associated block at time t (either because the nth block was withheld or overridden, or because $n > t$), we say that $B_t(n) = \emptyset$. Notice that due to adversarial manipulations, it is possible for $B_t(n) = \emptyset$ and $B_{t-1}(n) \neq \emptyset$, and vice versa.

In addition to the main chain, the adversary maintains arbitrarily many private *side chains*, $\tilde{B}_t^1, \ldots, \tilde{B}_t^s$, where s denotes the number of side chains. The blocks in each side chain must respect the global leader sequence $W(n)$. An adversary can choose at any time to publish a side chain, but we also assume that the adversary's attacks are *covert*: it never publishes a side chain that conclusively proves that it is keeping side chains. For example, if the main chain contains a block B created by the adversary for block slot n, the adversary will never publish a side chain containing block $\tilde{B} \neq B$, where $\tilde{B}$ is also associated with block slot n.

Each side chain $\tilde{B}_t^i$ with $i \in [s]$ overlaps with the honest chain in at least one block (the genesis block), and may diverge from the main chain after some $f_t^i \in \mathbb{N}_+$ (Figure 2.6). That is,

$$f_t^i := \max\{n \in \mathbb{N}_+ \,:\, B_t(n) = \tilde{B}_t^i(n)\}. \tag{2.36}$$

Different side chains can also share blocks; in reality, the union of side chains is a tree.

However, for simplicity of notation, we consider each path from the genesis block to a leaf of this forest as a separate side chain, instead of considering side trees. We use ℓ_t and $\tilde{\ell}_t^i$ to denote the chain length of B_t and $\tilde{B}_t^i$, respectively, at time t:

$$\ell_t = |\{n \in [T] \,:\, B_t(n) \neq \emptyset\}|\,, \quad \text{and} \quad \tilde{\ell}_t^i = |\{n \in [T] \,:\, \tilde{B}_t^i(n) \neq \emptyset\}|, \tag{2.37}$$

and we use the heights h_t and $\tilde{h}_t^i$ to denote the block indices of the ℓ_tth and $\tilde{\ell}_t^i$th blocks, respectively:

$$h_t = \max\{n \in [T] \,:\, B_t(n) \neq \emptyset\}\,, \quad \text{and} \quad \tilde{h}_t^i = \max\{n \in [T] \,:\, \tilde{B}_t^i(n) \neq \emptyset\}. \tag{2.38}$$

If $f_t^i = h_t$, then the adversary is building its ith side chain from the tip of the current main chain.

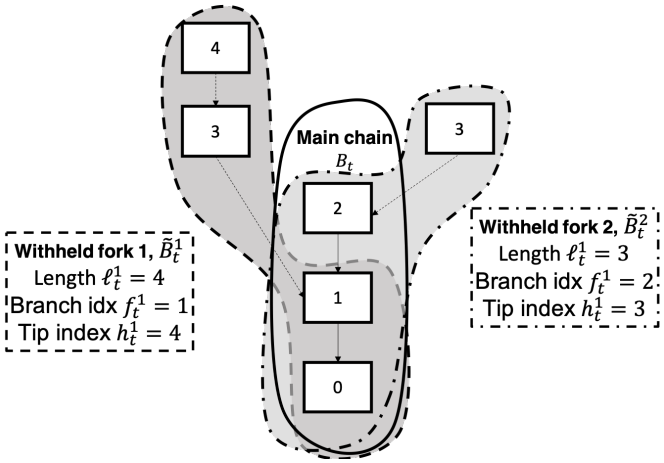


Figure 2.6: In PoS, the adversary can keep arbitrarily many side chains at negligible cost, and release (part of) a side chain whenever it chooses.

State space. The state space for the system consists of three pieces of data: (1) The current time $t \in [T]$; (2) The main chain B_t; and (3) The set of all side chains $\{\tilde{B}_t^i\}_{i\in[s]}$. Notice in particular that the set of side chains grows exponentially in t. In practice, most systems prevent the main chain from being overtaken by a longer side chain that branches more than Δ blocks prior to h_t, which is called a *long-range attack*. Hence we can upper bound the size of the side chain set by imposing the condition that for all $i \in [s]$, $h_t - f_t^i \leq \Delta$. However, the size of the state space is considerably larger than it is in prior work on selfish mining in PoW [55], where the computational cost of creating a block forces the adversary to keep a single side chain.

Objective. The adversary A's goal is to maximize its fraction of the total stake in the main chain by the end of the experiment,

$$v_A(t) \quad = \quad \frac{|\{n \in [T] \,:\, (W(n) = A) \wedge (B_T(n) \neq \emptyset)\}|}{\ell_T}. \tag{2.39}$$

This objective is closely related to the metric of prior work [55], except for the finite time duration.

Strategy space. The adversary has two primary mechanisms for achieving its objective: choosing where to append its blocks and when to release a side chain. If the honest party H is elected at time t, by the protocol, it always builds on the longest chain visible to it. As we assume small enough network latency, H appends to block $B_{t-1}(h_{t-1})$. However, if A is elected at time t, A can append to *any* known block in $B_{t-1} \cup \{\tilde{B}^i_{t-1}\}_{i\in[s]}$. The system must allow such a behavior for robustness reasons: even an honest proposer may not have received a block $B_{t-1}(h_{t-1})$ or its predecessors due to network latency.

The adversary can also choose when to release blocks. In our model, H always releases its block immediately when elected. However, an adversarial proposer elected at time t can choose to release its block at any time $\geq t$; it can also choose not to release a given block. Late block announcements are also tolerated because of network latency; it is impossible to distinguish between a node that releases their blocks late and a node whose blocks arrive late because of a poor network connection.

Notice that if A is elected at time t and chooses to withhold its block, the system advances to time $t+1$ without appending A's block to the main chain. This means that the next proposer $W(t+1)$ is selected based on the stake ratios at time $t-1$. So the adversary may have incurred a selfish mining gain from withholding its block, but it lost the opportunity to compound the t^{th} block reward. This tradeoff is the main difference between our analysis and prior work on selfish mining attacks in PoW systems.

Drawing from [40, 55], at each time slot t, the adversary has three classes of actions available to it: match, override, and wait.

1. The adversary **matches** by choosing a side chain $\tilde{B}^i_t$ and releasing the first h_t blocks. This means the released chain has the same height as the honest chain. In accordance with [40, 55], we assume that after a match, the honest chain will choose to build on the adversarial chain with probability γ, which captures how connected the adversarial party is to the rest of the nodes.
2. The adversary **overrides** by choosing a side chain $\tilde{B}^i_t$ and releasing the first $h = h_t + 1$ blocks. The released chain becomes the new honest chain.

3. If the adversary chooses to **wait**, it does not publish anything, and continues to build on all of its side chains.

Unlike [40, 55], we do not explicitly include an action wherein the adversary adopts the main chain. Because our model allows the adversary to keep an unbounded number of side chains, adopting the main chain is always a suboptimal strategy; it forces the adversary to throw away chains that could eventually overtake the main chain. The primary nuance in the adversary's strategy is choosing *when* to match or override (rather than waiting), and *which* side chain to choose. Identifying an optimal mining strategy through MDP solvers as in [55] is computationally intractable due to the substantially larger state space in this PoS problem.

2.5.1 Strategic selfish mining

We show that adversarial gains from strategic behavior are exacerbated by compounding. In practice, the adversary needs a strategy that balances the gains of keeping a long side chain to potentially overtake a long main chain, with the loss in intermediate leader elections due to withheld rewards. We propose a family of schemes called *Match-Override-k* (MO-k). Under MO-k, the adversary only keeps side chains whose tip is at most k blocks ahead of the main chain. The strategy is as follows: Every time a new honest block is generated, it is appended to the main chain. Next, if there is a side chain that (1) is longer than ℓ_t, and (2) does not already include the entire honest chain, the adversary *matches* the main chain. Now there are two chains of equal length in the system; with probability γ, the newly-released side chain becomes the new main chain. Otherwise, the previous honest main chain continues to be the main chain, and the failed side chain is discarded. If there is no such side chain to match, then the adversary *waits*. Any side chains shorter than ℓ_t are discarded.

Every time a new adversarial block is generated, the adversary appends it to every side chain it is managing currently. It also starts a new side chain branching from the tip of the main chain if there is not a side chain there already. The adversary now checks every side chain. If there is a side chain that branches at the tip of the main chain and is at least k blocks ahead of the main chain, the adversary *overrides* with this side chain, thereby incrementing the main chain length by one. Otherwise, the main chain remains as is, and the adversary *waits*.

Figure 2.7 simulates how much the adversary can gain in average fractional stake by using MO-k strategies. As the total reward R increases, the relative fractional stake approaches 3, which is the maximum achievable value, since the expected fractional stake is normalized by $v_A(0) = 1/3$. The simulations were run for $T = 10,000$ time steps, with $S(0) = 1$.

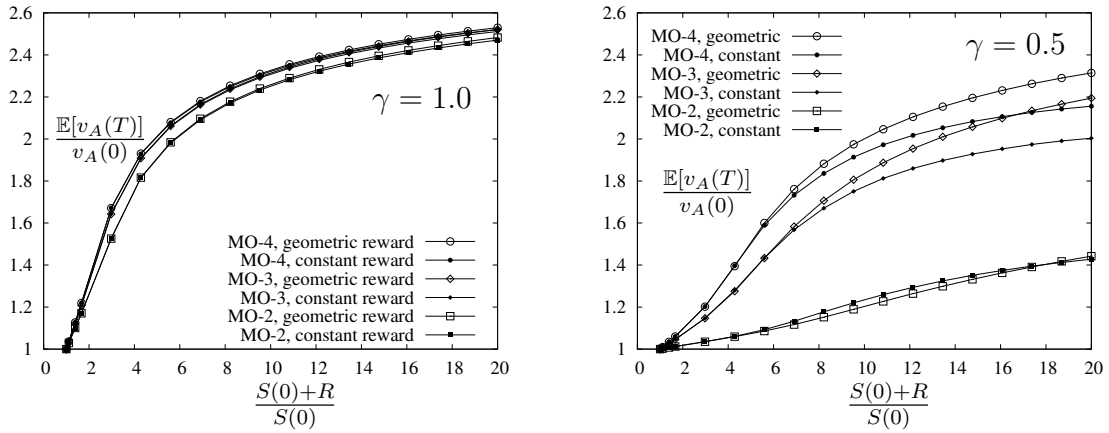


Figure 2.7: Average fractional stake of an adversary can increase significantly as the total reward R increases. We fix initial fraction $v_A(0) = 1/3$, $S(0) = 1$, and $T = 10,000$ time steps, and show for two values of network connectivity of the adversary $\gamma \in \{0.5, 1.0\}$ and varying total reward R.

When the adversary is well-connected, i.e., $\gamma = 1.0$, such attacks are effective even with short side chains, such as $k = 3$ or 4. Further, there is no distinguishable difference in the reward function used. On the other hand, when the adversary has 0.5 probability of matching honest chains, $\gamma = 0.5$, it is more effective to keep longer side chains. This figure demonstrates dramatic gains in fractional stake due to strategic behavior. A natural question is how large these gains can be. This question is sophisticated and discussed in the longer version of this chapter's work [57].

2.6 CONCLUSION

This work measures the concentration of wealth (token) in PoS systems, showing that existing block reward functions (e.g., constant) have poor equitability. We introduce a maximally-equitable geometric reward function. The negative effects of compounding can be further mitigated by choosing the total block rewards for each epoch to be small compared to the initial stake pool size.

Several open questions remain. First, our results do not account for scenarios where proposers add or remove stakes during an epoch. Another challenge, discussed in [57], is that geometric rewards may not be desirable in practice because of the sharp changes in block rewards between epochs. A natural solution is to impose smoothness constraints on the class of reward functions — an interesting direction for future work. Next, our analysis chooses the epoch length as the period between inflation checkpoints, usually a month or a year in practice. However, the epoch length is not necessarily related to such checkpoints.

A smaller epoch length such as a day or an hour leads to a different analysis. It is an interesting direction for future work to discuss which epoch length choice is more suitable for practice. Finally, although strategic players are not specific to PoS systems, we show that geometric rewards alone do not protect against them. Designing incentive-compatible consensus protocols is a major open question.

CHAPTER 3: REMOVING CONSENSUS BOTTLENECK FOR PERMISSIONLESS BLOCKCHAIN SYSTEMS

The performance of existing permissionless smart contract platforms such as Ethereum is limited by the consensus layer. Prism [6] is a new proof-of-work consensus protocol that provably achieves throughput and latency up to physical limits while retaining the strong security guarantees of the longest chain protocol. This chapter reports experimental results from implementations of two smart contract virtual machines, EVM and MoveVM, on top of Prism and demonstrates that the consensus bottleneck has been removed. Code can be found at `https://github.com/wgr523/prism-smart-contracts`.

We give an introduction to the work in §3.1. In §3.2 we discuss smart contract scaling approaches in different dimensions. §3.3 gives a brief overview of Prism consensus protocol. In §3.4, we describe our design and implementation of Prism with EVM and MoveVM. We present evaluation results of various canonical applications in EVM and MoveVM, and discuss their implications in §3.5. Conclusion is in §3.6.

This chapter is a joint work with Shuo Wang, Vivek Bagaria, David Tse, and Pramod Viswanath published as reference [58].

3.1 INTRODUCTION

Existing permissionless smart contract platforms such as Ethereum is based on the longest chain consensus protocol, the original blockchain protocol invented by Nakamoto [1]. While maintaining high security against adversarial attacks, it is well-known that the longest chain protocol suffers from poor throughput and latency performance. Hence, the performance of these platforms is limited by the consensus layer.

This limitation has led to practical congestion in the network. When CryptoKitties made its debut on Ethereum, a spike of transactions rushed into the system, far exceeding Ethereum's supported throughput. The pending transaction queue was growing quickly, and users had to increase transaction fees to incentivize miners to add their transactions to the chain. Decentralized Finance applications have been rapidly growing over the last few years and as it gets more popular in the near future, the demand will continue to grow, making the performance scaling of smart contract platforms an urgency.

Several promising efforts to scale the performance have been proposed. Almost every major live smart contract platform such as Ethereum, Algorand, and Tron are optimizing their existing smart contract engines to increase the throughput. A few others like Diem (led by Facebook) and Hyperledger Fabric (led by IBM) have taken the route of permissioned

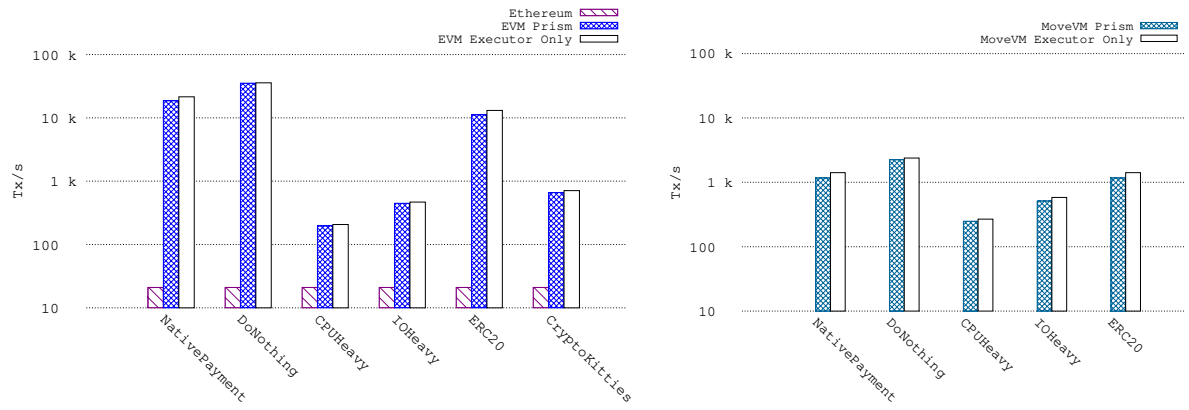
blockchains to obtain higher throughput. On the other hand, Ethereum foundation has taken a sharding approach to support higher throughput. Optimistic Rollup [59], ZK-Rollup [60], and Arbitrum [61] are other off-chain scaling solutions built on top of an existing smart contract platform such as Ethereum. In these off-chain solutions, not every validator node needs to keep track of the execution of the off-chain contracts, which leads to an improved overall efficacy but at the expense of security.

Prism [6] is a recent permissionless proof-of-work (PoW) consensus protocol which naturally scales the performance of the longest chain protocol. It provably achieves throughput and latency up to computation and communication limits of the underlying physical network, while retaining the strong security guarantees of the longest chain protocol. An implementation of Prism [62] scales performance significantly in a Bitcoin-like payment system, improving the throughput of Bitcoin by about 4 orders of magnitude. The question remains as to whether Prism can successfully support a general smart contract platform and remove the consensus bottleneck. Indeed, not every blockchain consensus protocol is extensible to a smart contract platform (e.g., Spectre [63]) and scalably integrating consensus with smart contract platforms is nontrivial.

This chapter demonstrates that Prism can support general smart contract platforms and provide a very high level of performance. We present the design and implementation of Prism that provides a flexible interface for connecting with two common smart contract virtual machines. We report experimental results from implementation of two smart contract virtual machines, Ethereum VM (EVM) and MoveVM, on top of Prism. Figure 3.1a shows throughput results for running several canonical smart contract applications on EVM on Prism, while Figure 3.1b shows analogous results for MoveVM on Prism. As can be seen, the throughputs are very close to that of virtual machine execution only *without consensus*, and much larger than the throughput using the longest chain protocol. Thus, we conclude that smart contract platforms built on Prism can perform without the consensus layer bottleneck.

3.2 RELATED WORK

The throughput of blockchains with smart contract platform can be increased at three different points on the blockchain stack. The first approach is to improve the execution speed of the virtual machine engine. A basic approach is to optimize the execution of individual op codes (followed in EVM clients such as Parity Ethereum and Geth) or by designing a new set of op codes from first principles (followed by Diem to arrive at MoveVM [64]). A more involved approach is to execute smart contracts in *parallel* similar to the modern design of databases such as MySql [65] and Postgres [66]. The first technique is to run multiple



(a) EVM Prism (b) MoveVM Prism

Figure 3.1: Throughput of Prism clients; experimented with 100 nodes on several applications: *Native Payment*, *Do Nothing*, and *ERC20* are lightweight applications whereas others are heavyweight ones. In MoveVM, *Native Payment* is essentially the same as *ERC20*. The reader is cautioned against comparing performance across the two VM's, as EVM is a mature technology while MoveVM is current under active development. Rather, the main point of obtaining results in two VM's is to demonstrate the flexibility of Prism. Moreover, we did not compare with the the performance of MoveVM on Diem consensus because it is permissioned while Prism is permissionless.

smart contracts in parallel where smart contracts acquire locks on a data before editing to ensure no data is simultaneously edited by more than a single smart contract. It is used in reference [67] with a 33% improvement in throughput. An alternative approach uses optimistic concurrency with rollbacks, where multiple smart contracts execute in parallel (without locks). When two smart contracts running in parallel try to edit the same data, one of them is rolled backed and executed later. This approach is explored in references [68, 69, 70, 71] where 3-4x improvement in throughput is observed. Although the improvement in throughput is significant in these methods, it exposes the blockchain to new kinds of adversarial attacks. Moreover these methods don't address metering which is a critical component to align incentives.

Even though the current VMs have low throughput, the current bottleneck in today's blockchain platform is the consensus protocol itself. The longest chain protocol and its current variants do not saturate the performance of the underlying VMs (refer Figure 3.1 for details). Therefore, the second approach of designing high throughput consensus protocols is a natural avenue to scale smart contract platforms. One method is to move from permissionless to permissioned consensus protocols which can support high throughput, and Facebook's Diem [72] and IBM's Hyperledger Fabric [30] take this path. Diem has chosen a recent high-performing Byzantine fault tolerance (BFT) consensus protocol (HotStuff [73]); Hyperledger Fabric [30] proposes the execute-order-validate paradigm in order to attain both

performance and extensibility, where (1) participants execute transactions and collect endorsements for the executions, (2) responsible participants order these executed transactions through a consensus protocol, and (3) transactions are validated by all participants. However, these approaches sacrifice the very important characteristic of being permissionless. In this chapter we take the approach of designing and implementing a high throughput *permissionless* consensus protocol, Prism, which achieves high throughput. Protocols such as OHIE [24], Algorand [21], and Bitcoin-ng [22] take a similar route. To the best of our knowledge, there do not exist implementations running smart contracts on top of these protocols; hence we have not been able to make a direct comparison with Prism's performance.

The third approach is Plasma and sharding. In 2015, Poon and Buterin proposed Plasma [74], an off-chain scaling solution. At a high level, Plasma is a network of secondary chains, and each one of them is designed to serve different needs. These chains interact not only among each other, but also with the main chain (on a need basis) to resolve conflicts using fraud proofs. This approach has weaker security properties and, in particular, susceptible to the "mass exit" attack. To overcome some of these security vulnerabilities, Ethereum 2.0 [75], near [76], polkadot [77], and Trifecta take the sharding approach which *horizontally* scales the throughput by running multiple instances of blockchains and pooling them to obtain high security. Even though this approach has better security than plasma, overall it has lower security compared to the pure consensus protocols in the previous paragraph.

3.3 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 3.2). 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.

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

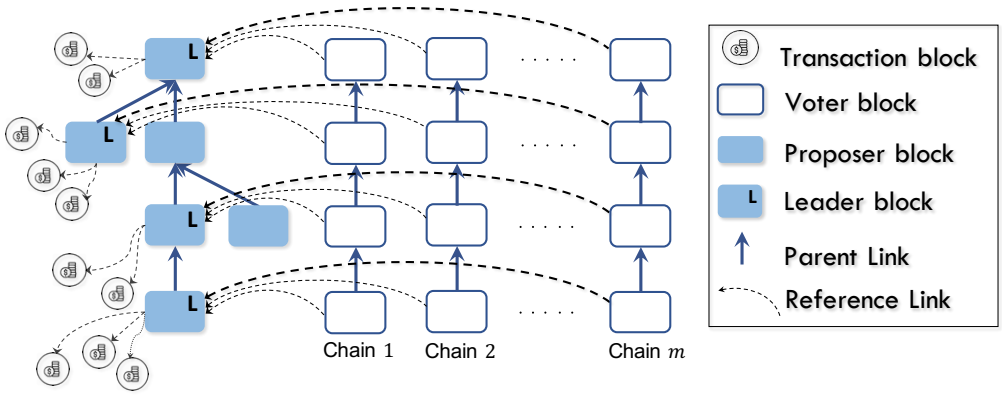


Figure 3.2: Prism: Factorizing the blocks into three types of blocks: proposer blocks, transaction blocks and voter blocks.

of proposer blocks, known as the *leader sequence*. We define the *level* of a proposer block as its distance from the genesis proposer block, and the *height* of the proposer tree as the maximum level that contains any proposer blocks. The leader sequence of proposer blocks contains one block at every level up to the height of the proposer tree, and is determined by the *voter chains*.

There are m voter chains, where $m \gg 1$ is a fixed parameter chosen by the system designer. 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 confirmed ledger.

By decoupling the various types of blocks, Prism can provably achieve low latency and high throughput while maintaining high security.

3.3.1 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.

Theorem 3.1 (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

$$Dc_1(\beta) + \frac{Dc_2(\beta)}{m} \log \frac{1}{\epsilon} \quad \text{seconds}, \tag{3.1}$$

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.

3.3.2 Throughput

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

To decouple security from throughput, transactions are instead carried by separate transaction blocks. Each proposer block when it is mined refers to the transaction blocks that have not been referred to by previous proposer blocks. This design allows throughput to be increased by increasing the mining rate of the transaction blocks, without affecting the security of the system. The throughput is only limited by the computing or communication bandwidth limit C of each node, thus potentially achieving 100% utilization.

Theorem 3.2 (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.

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

3.4 DESIGN AND IMPLEMENTATION

We implement a Prism full-node client with VMs in around 10,000 lines of Rust code. In this section, we describe the architecture of the client and highlight several design choices that are tailored to Prism consensus.

3.4.1 Architecture

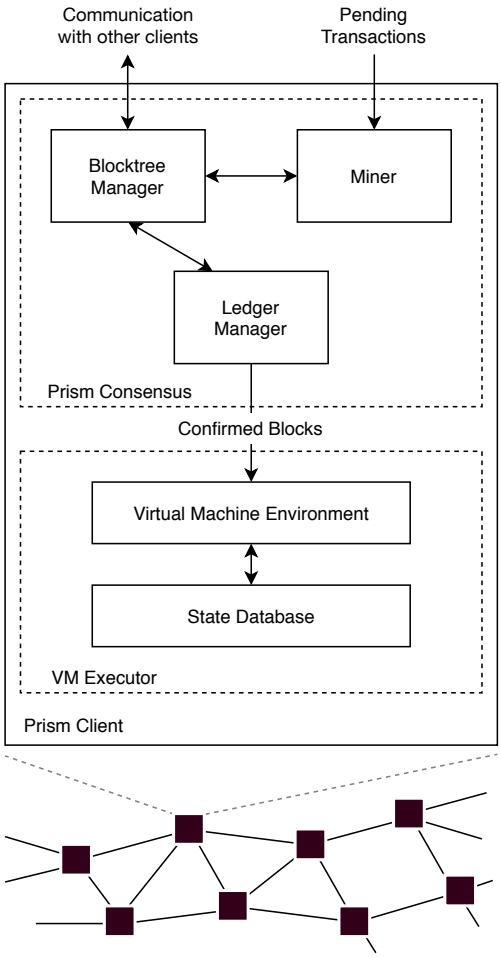


Figure 3.3: Architecture of the Prism client. In the peer-to-peer network, each node is running a Prism full-node client.

Our implementation of Prism full-node client consists of two modules, Prism Consensus module and Virtual Machine Executor (VM Executor) module. Prism Consensus module is in charge of exchanging blocks with peers, following Prism consensus to confirm blocks, and push confirmed blocks to VM Executor. VM Executor maintains the *state* of the confirmed ledger, i.e., the state that results from executing transactions up to the last confirmed block.

When VM Executor receives new confirmed blocks from Prism Consensus, it retrieves transactions from those blocks and updates the state accordingly. This architecture is illustrated in Figure 3.3.

Prism Consensus module can be divided into the following three parts:

1. Blocktree Manager, which maintains the client's view of the blockchain, and exchanges blocks with peers;
2. Ledger Manager, which confirms blocks by following Prism protocol, and pushes confirmed blocks to VM Executor;
3. Miner, which contains a transaction memory pool and assembles new blocks.

Blocktree Manager consists of an event loop and a thread pool. The event loop keeps listening to events such as sending/receiving blocks, and assigns a thread from the thread pool to process it. When the client receives a new block from a peer, Blocktree Manager checks its proof of work, and stores the block locally. After that, it relays the block to peers in case they have not received it. It then checks data availability, i.e., whether all the blocks referred by reference links in this block have been received. If not, it buffers the block and defers further processing until data availability is satisfied. After data availability is satisfied, Blocktree Manager checks sortition and transaction signatures. Finally the block is inserted into Prism blocktree.

Ledger Manager is a busy-waiting loop that queries Blocktree Manager periodically to see whether there are new confirmed blocks, following Prism's confirmation rule. If there are, it will retrieve the blocks from local storage and push them to VM Executor via a message-passing channel. The choice of the busy-waiting loop suits the high transaction workload since the busy-waiting overhead is negligible when it takes a long time to retrieve a large number of blocks and push them to VM Executor. Both Blocktree and Ledger Managers use RocksDB as the storage backend [78, 79] thanks to its high performance and ease of integration.

Miner module maintains a memory pool that collects pending transactions and assembles them into new blocks. The Miner module does not actually try to solve the PoW hash inequality. Instead, it simulates the mining process via a Poisson process (of fixed growth rate, corresponding to the mining difficulty level) which is statistically independent across different nodes (matching the distributed nature of PoW mining). When a new block is mined, it is pushed to Blocktree Manager which will broadcast the block to peers. Transactions carried by assembled or received blocks are checked for duplication in the memory pool, with duplicates being purged.

VM Executor is in charge of maintaining the *state database*, i.e., the persistent storage for the state of the confirmed ledger. State database stores account information such as

address and balance, and manage data in a hash accumulator (Merkle Patricia tree is used in Ethereum and sparse Merkle tree is used in Diem). VM Executor receives confirmed blocks from Ledger Manager, retrieves transactions from those blocks, and executes them sequentially. To execute a transaction, VM Executor first initializes a virtual machine environment, such as program counter, stack, and memory, and then it executes the instructions coded inside the transaction and/or the smart contract. During the execution, it may interact with the state database. The execution result of a transaction will be a success or a failure, depending on whether the transaction is valid or not. Invalid transactions with failure results should be *sanitized* out of the confirmed ledger and have no effect on the state. Valid transactions will update the state according to the execution result. After executing all transactions in a confirmed block, VM Executor commits the updates to the state database.

We ported the VM Executors from two open source projects, Open Ethereum [80] (popularly known as Parity Ethereum) and Diem [72], and adapt the structure of transactions, the hash function, and the signature schemes to these projects respectively. The port only required us to add or modify less than 20 lines of code (LOC) for Open Ethereum and less than 160 LOC for Diem in their Rust language codebases. In addition, 2 LOC were modified in Move language for Diem codebase. The two VM Executors run single threads, with no parallel transaction execution capability. We will use the name of virtual machines, EVM and MoveVM, to refer them hereafter.

3.4.2 Highlights

The key design and implementation challenge is in translating the high throughput, low latency and high confirmation probability that Prism provides on raw block and transaction level into an application layer programming construct via the virtual machine intermediaries. On the one hand, the client must process blocks and transactions at a rate much higher than most traditional blockchains. On the other hand, low latency and high confirmation probability enables confirmation of the ledger, which the implementation can benefit from. Here, we highlight several implementation choices that are tailored for Prism consensus and distinguish our implementation from traditional blockchains.

Confirmation. In Ethereum and other longest chain protocols, the state of the longest chain tip is used for transaction validation. However, blocks in longest chain may be switched due to honest or adversarial forking blocks. To smoothly update state when the longest chain switch happens, Ethereum's implementation keeps a short-term journal containing actions in recent forking blocks. It leads to the lower efficiency of the state management which is a

particular impediment due to the high mining rate (and high throughput) of Prism. In our design, we find it relevant to only maintain the state of the last confirmed block because of two reasons: (1) Prism guarantees confirmation with overwhelmingly high probability (e.g., $1 - 10^{-9}$), so confirmed blocks are not likely to be deconfirmed. (2) Prism does not validate transactions before including them in blocks, so it is unnecessary to maintain the state of the unconfirmed latest proposer block. It not only makes maintenance more efficient, but also enables the integration with VM of BFT consensus such as MoveVM.

In traditional blockchains (Bitcoin and Ethereum), blocks are mined at a relatively low rate and a newly mined block is likely to change the longest chain. Hence in their implementation, they update state when they receive a new block. In Prism, blocks are mined at a high mining rate; confirming blocks and updating state upon receipt of a new block would be onerous – we make a design choice to update the state only when blocks are confirmed and to conduct the confirmation procedure at regular intervals.

Decoupling Transaction Validation and State Update. In most traditional blockchains, transaction validation and state update are coupled with consensus. For example, Ethereum miners must make sure all the transactions in a block are valid, update Ethereum state accordingly, and record the result state root in that block. Prism, by design, decouples transaction validation and state update from consensus, that is, Prism miners do not conduct transaction validation or update state. Only after a block is confirmed, transactions in it are validated, and state is updated accordingly. In this procedure, invalid transactions are *sanitized* out of the confirmed ledger. We note that invalid transactions still incur gas fees for the senders and thus a rational user has no incentive to send invalid transactions. If the transaction sender has inadequate balance to pay the gas fee, the transaction will be treated as spam and skipped. Nevertheless this type of invalid transactions could reduce the utility of network bandwidth. To mitigate this spamming attack, miners could validate transactions (by checking sender's balance is no less than gas fee) with respect to their latest confirmed state, giving the adversary only a small window to create invalid transactions and spam the system. By this method, spam traffic is reduced by 80% whereas the confirmation latency is only increased by 5 seconds [62]. We do not implement this defense against spamming attack in this work.

Prior work Hyperledger Fabric [30] also separates transaction validation and state update from consensus by the following three-step execute-order-validate paradigm: (1) nodes execute transactions and collect endorsements for the executions, (2) responsible nodes order the executed transactions through a consensus protocol, and (3) the ordered transactions are validated by all nodes and the state is updated according to valid transactions. Prism

is similar to Hyperledger Fabric in that they both separate transaction validation and state update from consensus. Notably, a common feature for Prism and Hyperledger Fabric is that the ledger could possibly contain invalid transactions at first, which would be sanitized out of the ledger later. Nevertheless, they are different in two ways. (1) Prism orders transactions, then executes and validates them. In other words, it adopts the order-execute-validate paradigm in contrast to Hyperledger Fabric's execute-order-validate paradigm. This order-execute-validate paradigm is closely related to traditional consensus protocols, whereas the paradigm of Hyperledger Fabric deviates far from traditional ones. (2) In Hyperledger Fabric, the execution of a transaction only occurs on a special set of nodes, and the validation requires these nodes to sign and transmit endorsements. Whereas in Prism, validation does not require such endorsements in that a transaction is validated by checking its local execution outcome; this execution and validation are replicated on every node. These differences imply that Prism fits well with current platforms such as Ethereum and can replace traditional consensus protocols seamlessly, while Hyperledger Fabric requires some efforts to design full-node and light-node clients in order to meet its novel requirements.

No Pending Transaction Exchange. Most traditional blockchain clients exchange pending transactions in their memory pools with peers. Because the block mining rate is very low and the next block author is unpredictable, transaction exchange is necessary to ensure that pending transactions get included in the next block. This reduces network bandwidth utility since transactions are broadcast twice in the network: first as pending transactions and then as part of a block.

In Prism, pending transaction exchange can be onerous to the network bandwidth due to the high throughput. We design our implementation to avoid exchanging pending transactions by noting that a pending transaction can be easily included in a *new* block in a very short amount of time by any individual miner thanks to the high mining rate of Prism's transaction blocks. Transaction blocks carrying pending transactions are broadcast to peers, in the same way blocks are broadcast in traditional blockchains. Notice that a user can still send a transaction to multiple miners for redundancy. However, miners need not exchange it. Such a design avoids the waste of network bandwidth and contributes to the final high throughput.

Signature Verification in Consensus. Transaction signature verification is a significant fraction of total computation, and the computation becomes only heavier when the achieved throughput is higher. We design our implementation to conduct the signature verification in *parallel inside Prism Consensus* via the thread pool functionality. This is a departure

from implementations in EVM (Ethereum) and MoveVM (Diem) which conduct signature verification inside the VM executor. Either sequentially or in parallel, signature verification burdens the VM executor and harms the throughput.

3.5 EVALUATION

In this section we describe our experiments and performance results of our implementation of a prototype client designed by following the guidelines highlighted in the previous section. We describe experiment settings and the applications that we measure (§3.5.1). Then we present the throughput and confirmation latency results of Prism integrated with two virtual machines, EVM and MoveVM, from which we analyze that Prism removes the consensus bottleneck (§3.5.2, §3.5.3). In addition, we measure how our design and implementation of Prism scales with more network participants (§3.5.4).

3.5.1 Experiment Setting

We evaluate our implementation of Prism by integrating it with two smart contract virtual machines: *EVM Prism* and *MoveVM Prism* respectively. The performance (upper bound) baselines are provided by *VM Executor Only* (single node, no consensus) and *Prism Consensus Only* (no smart contract platform, raw transaction throughput). *VM Executor Only* experiment feeds transactions to VM Executor running on a single node and demonstrates the optimal throughput of the VM Executor. *Prism Consensus Only* experiment runs consensus with raw blocks and transactions and measures the raw data throughput. It shows the performance that the consensus is able to support. In addition, we also implement Ethereum's consensus protocol (essentially the longest chain protocol) and its performance provides a (lower bound) baseline.

Applications: We evaluate a suite of canonical applications and classify them into three categories.

1) *Basic applications:* We evaluate two basic applications: *Native Payment* and *Do Nothing*. *Native Payment* transactions are payments of native tokens in those smart contract platforms. *Do Nothing* is a contract with a void function, and is the simplest possible contract.

2) *Benchmark applications:* To test Prism client with standard computation or storage read/write, we propose two applications: *CPU Heavy* and *IO Heavy*. *CPU Heavy* runs a worst case of quick sort for an integer array of length 255. *IO Heavy* does key-value pair write 255 times followed by key-value pair read 255 times for both forward and backward

order (thus total 510 times). The value type is bytes32 in EVM and bytearray in MoveVM, which are both 256-bit data type.

3) *Realistic applications:* As a counterpoint to the above applications, we evaluate here the performance with respect to two real world applications: *ERC20* and *CryptoKitties*. ERC20 is an Ethereum token standard [81], and we implement it by using the reference implementation in [82]. CryptoKitties is a game that allows users to breed virtual pets. The genes of offspring are determined by a function named *mixGenes* that mixes the genes of its parents [83]. We adopt *mixGenes* function in our experiments, and feed random parent genes to it. This function is significantly computational heavy compared to basic applications.

Applications for EVM are developed in Solidity programming language. We use the official Solidity compiler v0.6.3 to compile all smart contracts to bytecode except for *CryptoKitties*, which we follow the version v0.4.18 in the contract. We set the compiler to Constantinople version and enable the default optimization. When creating a smart contract in EVM, an account address is created and bytecode is stored under the address. Applications for MoveVM are developed in Move IR. The smart contracts are first published as modules under the sender's address and then are called via scripts. We use Move IR compiler to compile the modules and scripts to bytecode. We have basic applications and benchmark applications and they have the same functionality as corresponding applications for EVM. Native tokens in MoveVM have essentially the same function as ERC20 tokens in EVM, hence *ERC20* experiment for MoveVM is unnecessary. As the Move language is in rapid development and not yet mature at the moment of our experiments, it is not straightforward to implement *CryptoKitties* in MoveVM.

Table 3.1 presents the statistics of applications. Transaction sizes differ because we pass different input parameters to these applications. Number of instruction and gas are indicators of the complexity in terms of both computation and storage read/write. MoveVM does not provide the statistics for number of instruction.

To generate the workloads for our evaluations, we implement a transaction generator that periodically generates transactions and push them into the mempool, generating different transaction types for different applications. We cap the generation rate according to the throughput of *VM Executor Only* experiment, in order not to exhaust the virtual machine.

We acquire data from the first 100 million transactions on Ethereum to derive a distribution on the number of transactions sent and received by an account. We sample our transactions using this distribution to mimic the usage of Ethereum in our experiments. For senders and receivers, a total of 10,000 accounts is used, respectively. The transaction gener-

[2]Since we pass random inputs to *CryptoKitties*, the number is also random and we present an approximation in the table.

Table 3.1: EVM and MoveVM application statistics.

	Native Payment	Do Nothing	CPU Heavy	IO Heavy	ERC20	Crypto-Kitties
EVM Tx Size (Bytes)	533	536	567	567	601	631
EVM Gas	21000	21394	334390	435244	26602	140000 [2]
Num of Instruction	0	32	88417	25364	309	25000 [2]
MoveVM Tx Size (Bytes)	424	329	365	366	N/A	N/A
MoveVM Gas	43076	629	2275420	2956846	N/A	N/A

ator of each node is initialized with 10,000 key pairs, one key pair for each account. In order to mimic the usage of Ethereum for *Native Payment* and *ERC20*, each node randomly and independently draws a sender and a receiver address from the aforementioned distribution. Other applications like *Do Nothing*, *CryptoKitties*, *CPU Heavy*, and *IO Heavy* have a fixed receiver (EVM) or no receiver (MoveVM) and hence we only sample the sender's address.

Experiment environment. We perform our experiments on 100 Amazon EC2 c5d.4xlarge instances. Each instance has 16 CPU cores, 32 GB memory, and NVMe SSD storage. Each instance hosts one Prism client and they are connected to form a random 4-regular topology, the diameter of which is 6. To emulate a realistic peer-to-peer network, we introduce a propagation delay of 120 ms on each link to match the typical delay in Ethereum's network [84], and a rate limiter of 300 Mbps for both ingress and egress traffic except for *Prism Consensus Only* experiment where the rate limiter is 600 Mbps in order to show the performance upper bound that the consensus can reach.

Parameters. For *EVM Prism* and *MoveVM Prism*, we choose a high adversarial hash power capability of $\beta = 0.4$ and a very low deconfirmation probability $\epsilon = 2 \times 10^{-9}$. We use $m = 1000$ voter chains and cap the size of transaction blocks to be 200 tx/block. Given the testbed with 120 ms peer-to-peer delay, we tune the mining rate of Prism's proposer and voter blocks to be 0.08 block/s, at which the empirical forking rate [3] is less than 0.11 in all experiments, and thus it ensures the security of Prism. We tune the mining rate of transaction blocks differently for different applications to match the throughput of *VM Executor Only* experiment: In *EVM Prism*, *Native Payment* 108; *Do Nothing* 180; *ERC20* 70; *CryptoKitties* 3.78; *CPU Heavy* 1.08; *IO Heavy* 2.34 block/s. In *MoveVM Prism*, *Native Payment* 12.6; *Do Nothing* 7.2; *CPU Heavy* 1.44; *IO Heavy* 3.06 block/s.

For the *Prism Consensus Only* experiment, we increase the size of transaction blocks to

[3] Forking rate is calculated by $1 - \frac{\#\text{ blocks in longest chain}}{\#\text{ blocks}}$.

400 tx/block and the mining rate to 200 block/s in order to show the performance upper bound that the consensus can reach. As for the *Ethereum* experiment, we use a mining rate of 0.1 block/s and a block size of 200 tx/block, which resemble the live Ethereum parameters.

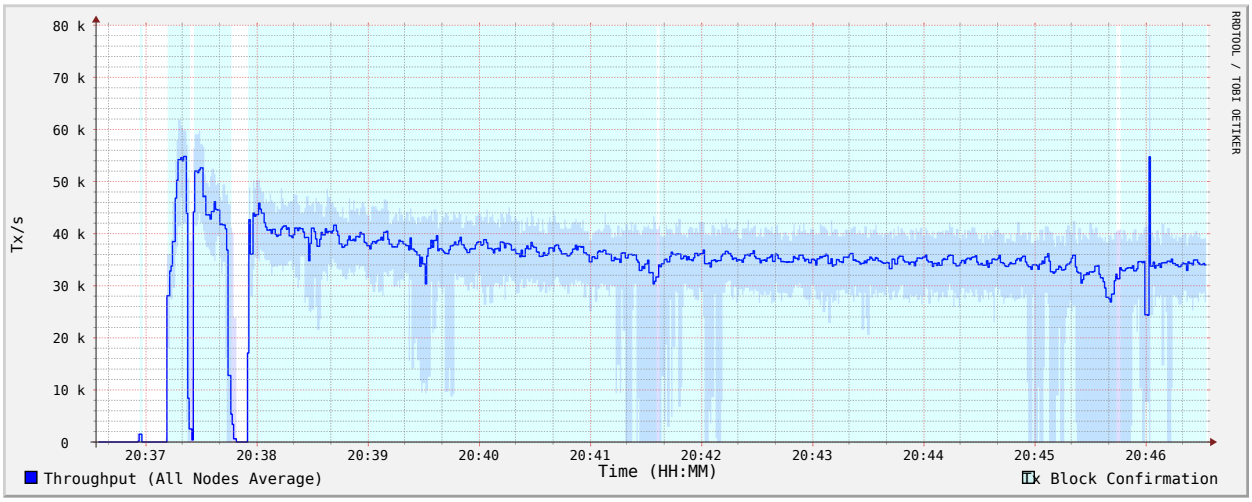


Figure 3.4: Time series plot of the throughput for *Do Nothing* application in *EVM Prism* for 10 minutes. Around the first 30 seconds there are very few processed transactions, since the clients are just started and have not extended the ledger significantly. After the ledger starts to be extended significantly, the throughput soon increases and becomes stable. This phenomenon occurs in all Prism experiments.

All experiments are run for at least 10 minutes. As we see in the time series plot of the throughput (Figure 3.4), in the first several seconds, the nodes don't process any transaction because they just started mining blocks and there are not enough blocks to extend the confirmed ledger. This phenomenon only happens at the beginning and does not affect the performance afterwards. Hence, the final throughput calculation involves the average performance over the last 9 minutes of the experiment.

3.5.2 Throughput and Latency of EVM

In this experiment, we measure the transaction throughput and confirmation latency of various applications in *EVM Prism* and analyze the difference in throughput for different applications. We also compare the throughput with *EVM Executor Only* experiment, the optimal throughput of EVM on a single node. If the former is able to reach the latter, then we will know the throughput of our Prism client comes very close to the optimal one of the virtual machine. And we can conclude that Prism removes the consensus bottleneck for smart contracts. Finally we compare *EVM Prism* with *Prism Consensus Only* to study whether Prism is able to support even higher throughput without the limitation of the virtual

machine. This experiment would also indicate whether *EVM Prism*'s performance can be further improved if the underlying virtual machine becomes faster.

Table 3.2: Throughput in terms of tx/s on EVM applications.

	Native Payment	Do Nothing	CPU Heavy	IO Heavy	ERC20	Crypto-Kitties
EVM Executor Only	21535	35723	207	467	13095	710
EVM Prism	18660	35329	197	447	11210	661
Prism Consensus Only [4]	98022	97473	92144	92144	86931	82798
Ethereum	21					

Throughput: As shown in Table 3.2, for *EVM Prism*, the throughput of two basic applications is able to reach 18K and 35K tx/s respectively. For *ERC20*, *EVM Prism* gets 11K tx/s. The throughput of these three applications shows that we have a good chance to get above ten thousand tx/s for those applications that do not involve heavy computation or storage read/write. The reason that *Do Nothing* is almost twice as fast as *Native Payment* is that for *Do Nothing*, the VM Executor module updates the account information of a random sender per transaction and a fixed receiver contract account, whereas for *Native Payment* it updates a random sender and a random receiver account information. As a result, the VM Executor needs to maintain the state database and hash accumulator for half account information updates in *Do Nothing* as that in *Native Payment*.

For the *CryptoKitties* application, *EVM Prism* achieves 661 tx/s due to its computational heavy nature. Similar things happen for *CPU Heavy* and *IO Heavy*, which get 197 and 447 tx/s respectively. According to the statistics in Table 3.1, these applications require more than 25K instructions in the virtual machine, which explains their low throughput. However, the low throughput in both *EVM Prism* and *EVM Executor Only* also indicates that EVM has a large opportunity to improve the efficiency of execution. We write exactly the same *CPU Heavy* application in Java and run in JVM, and we get a throughput over 90K tx/s. Considering the large gap between 197 and 90K, we believe that EVM has the potential to execute instructions more efficiently. We don't compare *IO Heavy* or *CryptoKitties* since they are not as straightforward to implement as a standalone program in Java.

Is Prism consensus the bottleneck? For all EVM applications, *EVM Prism* reaches 85% of *EVM Executor Only* throughput. This high percentage indicates that *EVM Prism* is able to reach the optimal EVM throughput very closely. As for *Prism Consensus Only*, we

[4]In *Prism Consensus Only*, the consensus throughput is 418 Mbps and is converted to tx/s based on transaction size.

can see the high throughput over 80K tx/s for all applications. This high number illustrates the ability of supporting a high throughput without the limitation of the virtual machine. It also shows that if the virtual machine becomes faster in the future, Prism is able to support its performance as well. Hence, Prism consensus is not the throughput bottleneck; the virtual machine itself is the bottleneck.

Compared to 21 tx/s in *Ethereum* experiment, which adopts Nakamoto's longest chain consensus, it is clear that the current Ethereum is limited by consensus.

Latency: The end to end latency of a transaction consists of two parts: confirmation latency and execution latency. Confirmation latency is the time between the generation of a transaction and the confirmation of the corresponding block. This latency is decided by Prism's confirmation rule and has a proved bound [6]. Execution latency is the time of waiting in a queue to be executed and execution combined. As long as we cap the transaction generation rate below the optimal virtual machine throughput in experiments, the time in the queue is negligible. Also the execution time is less than ten milliseconds since all applications have over one hundred tx/s throughput. Hence, execution latency is negligible compared to confirmation latency.

Prism's confirmation rule guarantees a confirmation latency regardless of its throughput. In all Prism experiments including *EVM Prism*, *MoveVM Prism*, and *Prism Consensus Only*, the confirmation latency is no more than 130 seconds. Notice that this latency is achieved with adversarial ratio $\beta = 0.4$ and reversal probability $\epsilon = 2 \times 10^{-9}$. To provide the same latency under the same condition in Ethereum, it needs to wait for ($k = 267$)-deep [1] and it translates to 2670 seconds if a block is mined in 10 seconds on average.

Resource utility: In a Prism client, the Prism Consensus module uses multiple threads to process messages from/to peers efficiently. The VM Executor module, on the contrary, runs in a single thread. In addition, RocksDB uses a few threads in the background. In total, a Prism client should only use no more than 50 threads. In our experiments, the live usage of CPU never exceeds 50% per core on average (notice that one instance has 16 CPU cores). It is likely that we can optimize CPU usage in the future, for example, give a high priority to the VM Executor thread to prevent it from competing with the Prism Consensus module for CPUs.

By profiling the CPU usage of a client in *EVM Prism Do Nothing* experiment, we find that transaction signature verification takes up to 39.2% of total CPU time (excluding mining), a relatively high percentage. When the experiment is running at a high throughput, the requirement of a large amount of signature verification becomes a major bottleneck, highlighting the importance of removing signature verification from the VM Executor module. In our design, we have moved signature verification into the Prism Consensus module and

set the VM Executor free from its heavy burden.

The VM Executor of EVM is implemented efficiently with abundant number of in-memory cache. However, it levies a heavy memory burden on the VM Executor. As our port of EVM does not include the whole client-level cache management, the VM Executor does not free memory efficiently. On the contrary, the memory usage increases along with the workload. Possibly, future optimization is needed for our port of EVM.

Table 3.3 provides a breakdown statistics for three Prism block types in *EVM Prism Do Nothing* experiment. We can see that transaction blocks take up to 71.2% of total generated block data, other two blocks only 28.8%. This indicates that the majority of utilized bandwidth contributes to the high throughput (transaction blocks), whereas Prism overhead takes up only a small fraction (proposer and voter blocks). For other *EVM Prism* (and *MoveVM Prism*) experiments, this breakdown statistics will remain similar except for transaction blocks. The higher the throughput, the higher the transaction block data and percentage. Thus, we do not analyze the breakdown statistics for other experiments.

Table 3.3: Statistics for three block types in *EVM Prism Do Nothing* experiment in 10 minute duration.

	# Mined Block	Block Data	Data Percentage
Proposer Block	44	4.0 MB	4.0%
Voter Block	47166	25.2 MB	24.8%
Transaction Block	107514	72.2 MB	71.2%

3.5.3 Throughput and Latency of MoveVM

In this experiment, we measure the transaction throughput and confirmation latency for *MoveVM Prism*. We observe similar bottleneck and latency between this experiment and EVM experiment, whereas there are also discrepancies in terms of throughput.

Throughput: As shown in Table 3.4, the throughput of two basic applications is only 1.1K and 2.2K tx/s respectively, which is an order of magnitude smaller than that of *EVM Prism*. In private communication [85], core Diem developers have indicated to us that improving the performance of MoveVM is in progress — when this improvement transpires, our Prism client can fully utilize that as well. Benchmark applications get 249 and 512 tx/s and are higher than those of *EVM Prism*, indicating that MoveVM is more efficient at executing instructions. The *CPU Heavy* throughput number, however, is still far below that of JVM (over 90K tx/s), so we believe MoveVM has the potential to execute instructions

more efficiently as well.

Table 3.4: Throughput in terms of tx/s on MoveVM applications.

	Native Payment	Do Nothing	CPU Heavy	IO Heavy
MoveVM Executor Only	1441	2501	269	585
MoveVM Prism	1172	2243	249	512
Prism Consensus Only [4]	123222	158802	143140	142749

Is Prism consensus the bottleneck? For all MoveVM applications, *MoveVM Prism* reaches 81% of *MoveVM Executor Only* throughput. This phenomenon is similar to EVM and indicates that *MoveVM Prism* is able to reach the optimal MoveVM throughput. As for *Prism Consensus Only*, we can see the high throughput over 120K tx/s as well. Similar to the case of EVM, we conclude that Prism removes the consensus bottleneck for MoveVM, and the virtual machine itself is the bottleneck.

Latency: Prism guarantees a confirmation latency regardless of the throughput, and we do observe that in all Prism experiments including *MoveVM Prism*, the average confirmation latency is no more than 130 seconds.

Resource utility: *MoveVM Prism* maintains a good memory usage which is kept under 3.2 GB in all experiments. The live usage of CPU never exceeds 32% per core on average. Compared to *EVM Prism* experiment, this CPU utility reduction is due to smaller throughput and more efficient signature verification. MoveVM adopts Ed25519 signature [86] which is faster than ECDSA [87] adopted by EVM.

3.5.4 Scalability

In this experiment, we evaluate Prism's ability to scale with more network participants. We use a larger number, 300, EC2 instances and the same propagation delay and rate limiter. We use a random 5-regular topology for 300 nodes, keeping diameter the same with that of 100 nodes. We also keep the same Prism parameter, including the overall mining rate, so that the individual mining rate is modified. By our design, only the Prism Consensus module is related to scaling with more network participants as it is in charge of communicating with peers. In addition, Prism Consensus module's performance is not affected by which application it is running. Hence, it suffices to experiment with one VM and application to demonstrate Prism's scalability. In the experiment, we use *EVM Prism* and *Native Payment*.

The experiment for 300 nodes also runs for 10 minutes. However, it is hard to collect the

fine-grained metrics for such a high number of nodes. So we calculate the overall metrics at the end of the experiment (all 10 minutes), in contrast to previous calculation (last 9 minutes).

Table 3.5 compares the performance between 100 and 300 nodes. The throughput and latency are very similar; the difference is due to the randomness of the experiments. The forking rate 0.113 in 300 nodes is a little larger than that in 100 nodes due to more hops and higher delay to propagate blocks throughout the peer-to-peer network, as we can see that the average path length is higher in the 300-node topology. This forking rate 0.113 is small enough to ensure the security of Prism consensus as well. In addition, the block propagation delay, as well as the forking rate, can be reduced by increasing the degree (the number of peers per node) of the peer-to-peer network; Geth [88] and Parity Ethereum [80] client have a default maximum degree of 50, which can sustain a low forking rate for peer-to-peer networks with a larger number of nodes.

Table 3.5: Performance of *EVM Prism Native Payment*, with different network topologies.

#Node	Degree	Average Path Length	Diameter	Throughput	Confirmation Latency (s)	Forking
100	4	3.55	6	17268	96	0.102
300	5	3.84	6	17417	80	0.113

Resource utility on each node is also similar. In the 300-node experiment, the live usage of CPU never exceeds 50% per core on average. The heavy memory burden of the VM Executor module is also similar to that in 100-node experiment.

We conclude that Prism is able to scale to a large number of network participants, as long as the underlying peer-to-peer network provides a topology with reasonable block propagation delay. We can achieve similar throughput, latency, and security in those cases.

3.6 CONCLUSION

Blockchain research thus far has progressed in a compartmentalized manner: algorithms and protocols (many focused on consensus) are designed and studied separately from the upper layer wrappers (virtual machine, application programming) they will interact with. This is in contrast with Nakamoto's Bitcoin design that was envisioned and designed as a complete system. This layering philosophy works well when the consensus layer is the bottleneck and much work can be expended to improve the performance (indeed, this is the

case with many blockchains, including Ethereum). Prism is a recent consensus algorithm, closely inspired by Nakamoto's longest chain protocol, with *theoretically optimal* throughput and latency. In this chapter we explore how Prism fits with two smart contract virtual machines, EVM and MoveVM, by implementing Prism underneath these virtual machines. We demonstrate that Prism seamlessly merges with both these VMs: our implementation approaches the *optimal* virtual machine throughput for a large variety of applications. This result means that Prism removes not only the consensus bottleneck of bare metal throughput and latency, but also that when interacting with two popular smart contract platforms. Further improvement of the smart contract performance would have to come from new designs of virtual machines and compilers and architectures capable of parallel execution of smart contracts. The early research in this area [67, 69] now takes on added urgency.

To think one step ahead, after both the consensus bottleneck and smart contract bottleneck are removed, what will be the bottleneck then? In reference [62], the Prism implementation has an efficient transaction execution module for the UTXO model, and its experiments show empirical evidence that improvements in database (e.g., tuning the performance) and storage device (e.g., using better NVMe SSD) push the transaction throughput higher. Therefore, a highly possible answer to the above question is that the storage of blockchains will be the bottleneck after the removal of consensus and smart contract bottlenecks. Research on blockchain storage is of importance under this scenario.

CHAPTER 4: BLOCKCHAIN CONSENSUS PROTOCOL FORENSICS

Blockchain consensus protocols focus on Byzantine fault-tolerant (BFT) consensus, where a group of replicas come to a consensus even when some of the replicas are Byzantine faulty. Multiple BFT protocols exist to securely tolerate an optimal number of faults t under different network settings. However, if the number of faults f exceeds t, security could be violated. In this chapter, we mathematically formalize the study of *forensic support* of BFT protocols: we aim to identify as many malicious replicas as possible and in an as distributed manner as possible. Our main result is that forensic support of BFT protocols depends heavily on minor implementation details that do not affect the protocol's security or complexity. Focusing on popular BFT protocols (PBFT, HotStuff, VABA, Algorand), we characterize their forensic support precisely, showing that there exist minor variants of each protocol for which the forensic supports vary widely. We show strong forensic support capability of DiemBFT, the consensus protocol of Diem cryptocurrency. Furthermore, our lightweight forensic module implemented on a Diem client is open-sourced [11] and under active consideration for deployment in Diem. Finally, we show that all secure BFT protocols designed for $2t + 1$ replicas communicating over a synchronous network forensic support are inherently nonexistent. This impossibility result holds for all BFT protocols even if one has access to all states of all replicas (including Byzantine ones).

We give an introduction of the problem in §4.1. We describe our results in the context of related work in §4.2. The formal problem statement and the security model is in §4.3. The forensic support of PBFT, HotStuff, VABA, and Algorand are explored in §4.4, §4.5, §4.6, §4.7, respectively. The forensic study of DiemBFT and the implementation of the corresponding forensic protocol is in §4.8. The impossibility of forensic support for all BFT protocols operating in the classical $n = 2t + 1$ synchronous network setting is shown in §4.9. Discussion of the results across 5 protocols studied is in §4.10.

This chapter is a joint work with Peiyao Sheng, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath published as reference [89].

4.1 INTRODUCTION

The core theoretical security guarantee of BFT consensus protocols is that as long as a certain fraction of nodes are "honest", i.e., they are non-faulty and follow the protocol, then these nodes achieve consensus with respect to (a time-evolving) state machine regardless of the Byzantine actions of the remaining malicious (or faulty) nodes. When the malicious

Table 4.1: Summary of notations.

Symbol	Interpretation
n	total number of replicas
t	maximum number of faults for obtaining agreement and termination
f	actual number of faults
m	maximum number of Byzantine replicas under which forensic support can be provided
k	the number of different honest replicas' transcripts needed to guarantee a proof of culpability
d	the number of Byzantine replicas that can be held culpable in case of an agreement violation

nodes are sufficiently numerous, e.g., strictly more than a 1/3 fraction of nodes in a partially synchronous network, they can "break security", i.e., create different views of the state machine at the honest participants.

In this chapter, we focus on events *after* malicious nodes have successfully mounted a security breach. Specifically, we focus on identifying which of the participating nodes acted maliciously; we refer to this action as "forensic support" and it should meet the following two goals:

- identify as many as possible the nodes that acted maliciously with an irrefutable cryptographic proof of culpability;
- identification is conducted as distributedly as possible, e.g., by the individual nodes themselves, with no/limited communication between each other after the security breach.

Our central finding is that the forensic possibilities crucially depend on minor implementation details of BFT protocols which do not affect protocol security or performance (latency and communication complexity). Our findings are demonstrated in the context of several popular BFT protocols for Byzantine Agreement (BA). We mathematically formulate "forensic support" of BFT protocols to measure the forensic abilities. The forensic support is parameterized as a triplet, (m, k, d), that represents the aforementioned goals. The triplet, (m, k, d), along with traditional BFT protocol parameters of (n, t, f) is summarized in Table 4.1. We emphasize that each of the protocol variants is safe and live when $f \leq t$ (here $n = 3t + 1$ for all the protocols considered), but their forensic supports are quite different.

Security attacks on BFT protocols can be far more subtle than simply double voting, and the forensic analysis is correspondingly subtle. We have analyzed the forensic support of

Table 4.2: Summary of results; the forensic support values of d are the largest possible and $n = 3t + 1$ here.

Protocols	Forensic Support	Parameters		
		m	k	d
PBFT-PK HotStuff-view VABA	Strong	$2t$	1	$t+1$
HotStuff-hash	Medium	$2t$	$t+1$	$t+1$
PBFT-MAC	None	$t+1$	$2t$	0
HotStuff-null				1
Algorand				0

classical-style BFT protocols, and our main findings are summarized in Table 4.2.

- **Parameter d.** We show that the number of culpable nodes d that can be identified is either 0 or as large as $t+1$. In other words, if at least one replica can be identified, we can also identify the largest possible number, $t+1$. The only exception is for HotStuff-null variant, where in a successful safety attack, we can identity the culpability of one malicious replica.
- **Parameter m.** We show that the maximum number of Byzantine replicas m allowed for nontrivial forensic support (i.e., $d > 0$) cannot be more than $2t$. Furthermore, any forensic support feasible with m is also feasible with m being its largest value $2t$, i.e., if (m, k, d) forensic support is feasible, $(2t, k, d)$ is also feasible.
- **Parameter k.** At least one replica's transcript is needed for forensic analysis, so $k = 1$ is the least possible value. This suffices for several of the BFT protocol variants. However, for HotStuff-hash variant, k needs to be at least $t + 1$ for any nontrivial forensic analysis.
- **Strong forensic support.** The first three items above imply that the strongest possible forensic support is $(2t, 1, t+1)$. Further, the BFT protocols in Table 4.2 that achieve any nontrivial forensic support automatically achieve the strongest possible forensics (the only exception is HotStuff-hash, for which the forensic support we identified is the best possible).
- **Impossibility.** For certain variants of BFT protocols (PBFT-MAC, HotStuff-null, and Algorand), even with transcripts from all honest replicas, non-trivial forensics is simply not possible, i.e., $d = 0$ even if m is set to its smallest and k set to its largest possible values ($t+1$ and $2t$ respectively); again, HotStuff-null allows the culpability of a single malicious replica.
- **Practical impact.** As forensic support is of immense importance to practical blockchain

systems, we conduct a forensic support analysis of DiemBFT, the consensus protocol in the new cryptocurrency Diem, which shows in-built strong forensic support. We have implemented the corresponding forensic analysis algorithm inside of a Diem client and built an associated forensics dashboard. Our reference implementation is available open-source [11] and under active consideration for deployment in Diem.

- **BFT with $n = 2t + 1$.** For a secure BFT protocol operating on a synchronous network, the ideal setting is $n = 2t + 1$. For *every* such protocol we show that at most one culpable replica can be identified (i.e., d is at most 1) even if we have access to the state of all honest nodes, i.e, $k = t$.

4.2 RELATED WORK

BFT protocols. PBFT [7, 8] is the first practical BFT SMR protocol in the partially synchronous setting, with quadratic communication complexity of view change. HotStuff [2] is the first partially synchronous SMR protocol that enjoys both a linear communication of view change and optimistic responsiveness. Validated Asynchronous Byzantine Agreement is solved by a state-of-the-art work [3] with asymptotically optimal communication complexity and round number. Algorand [9, 21] designs a committee self-selection mechanism, and the Byzantine Agreement protocol run by the committee decides the output for all replicas.

Usually SMR protocols have the corresponding Byzantine Agreement (BA) counterparts, due to the fact that these SMR protocols reach agreements on a log of values in a value-by-value way, so it is feasible to convert SMR ones to BA ones, also called *single-shot* BFT protocols. In the lens of blockchain, some SMR protocols have pipelined the phases to decide values, and are called *chained* BFT protocols, e.g., chained HotStuff [2] and DiemBFT [10] (formerly known as LibraBFT).

Beyond one-third faults. The seminal work of [5] shows that it is impossible to solve BA when the adversary corrupts one-third replicas for partially synchronous communication (the same bound holds for SMR). In BFT2F [90], a weaker notion of safety is defined, and a protocol is proposed such that when the adversary corrupts more than one-third replicas, the weaker notion of safety remains secure whereas the original safety might be violated. However, the weaker notion of safety does not protect the system against common attacks, e.g., double-spending attack in distributed payment systems. Flexible BFT [91] considers the case where clients have different beliefs in the number of faulty replicas and can act to confirm accordingly. Its protocol works when the sum of Byzantine faults and alive-but-corrupt faults, a newly defined type of faults, are beyond one-third. Two recent works [92, 93]

propose BFT SMR protocols that can tolerate more than one-third Byzantine faults after some specific optimistic period. The goal of these works is to mask the effects of faults, even when they are beyond one-third, quite different from the goals of forensic analysis.

Distributed system forensics. Accountability has been discussed in seminal works [94, 95] for distributed systems in general. In the lens of BFT consensus protocols, accountability is defined as the ability for a replica to prove the culpability of a certain number of Byzantine replicas in [96]. Polygraph, a new BFT protocol with high communication complexity $O(n^4)$, is shown to attain this property in [96]. Reference [97] extends Polygraph to an SMR protocol, and devises a finality layer to "merge" the disagreement. Finality and accountability are also discussed in other recent works, including Casper [46], GRANDPA [98], and Ebb-and-flow [99]. Casper and GRANDPA identify accountability as a central problem and design their consensus protocols around this goal. Ebb-and-flow [99] observes that accountability is immediate in BFT protocols for safety violations via equivocating votes; however, as pointed out in [96], safety violations can happen during the view change process and this is the step where accountability is far more subtle.

Reference [96] argues that PBFT is not accountable and cannot be modified to be accountable without significant change/cost. We point out that the definition of accountability in [96] is rather narrow: two replicas with differing views must be able to identify culpability of malicious replicas by themselves. On the other hand, in forensic support, we study the number of honest replicas (not necessarily restricted to the specific two replicas which have identified a security breach) that can identify the culpable malicious replicas. Thus the definition of forensic support is more flexible than that of accountability. Moreover, our work shows that we can achieve forensic support for protocols such as PBFT and HotStuff without incurring additional communication complexity other than (i) sending a proof of culpability to the client in case of a safety violation, and (ii) the need to use aggregate signatures instead of threshold signatures.

Relationship with MPC. In the setting of secure multiparty computation (MPC), the malicious adversary model specifies that adversaries may deviate arbitrarily from the protocol, which shares the same characteristic as Byzantine adversaries in consensus. Reference [100] introduces covert security model where the adversaries are willing to behave maliciously only if they are not caught. Furthermore, reference [101] proposes public verifiable covert (PVC) security model which adds the restriction that the detection of malicious adversaries should be publicly verifiable. In the model for forensic support, the definition of "irrefutable proof" has the same idea as public verifiable detection in PVC model. In both

models, the detection of adversaries is required to persuade other parties of the adversaries' identify. In addition, malicious parties cannot forge such a proof in the attempt of framing honest ones.

4.3 PROBLEM STATEMENT AND MODEL

The goal of SMR is to build a replicated service that takes requests from clients and provides them with the interface of a single non-faulty node, i.e., each client receives the same totally ordered sequence of values. To achieve this, the replicated service uses multiple nodes, also called replicas, some of which may be Byzantine, where a faulty replica can behave arbitrarily. Replicas finalize a prefix of the value sequence and output it to clients. A secure state machine replication protocol satisfies two guarantees. **Safety**: Any two honest replicas cannot output different sequences of values. **Liveness**: A value sent by a client will eventually be output by honest replicas.

SMR setting also has external validity, i.e., replicas only output non-duplicated values sent by clients. These values are eventually learned by the clients. Depending on the context, a replica may be interested in learning about outputs too. Hence, whenever we refer to a client for learning purposes, it can be an external entity or a replica. A table of notations is in Table 4.1. The formal definition of validated SMR is as follows:

Definition 4.1 (Validated State Machine Replication)**.** A protocol solves validated state machine replication among n replicas tolerating a maximum of t faults, if it satisfies the following properties:

(Safety) Any two honest replicas output two sequences s and s', then one is the prefix of the other, i.e., $s \preceq s'$ or $s' \preceq s$.

(Validity) If an honest replica outputs a sequence that contains a value v, v is an externally valid value, i.e., v is signed by a client.

(Liveness) An externally valid value v sent by a client will eventually be output by honest replicas.

This chapter mainly focuses on the setting of outputting a *single value* instead of a sequence of values for simplicity, since it is not hard to extend our results from single-shot consensus protocols to SMR ones. The problem is called Byzantine agreement (BA), and the safety and liveness properties of BA can be expressed using the following definition:

Definition 4.2 (Validated Byzantine Agreement)**.** A protocol solves validated Byzantine agreement among n replicas tolerating a maximum of t faults, if it satisfies the following properties:

(Agreement) Any two honest replicas output values v and v', then $v = v'$.

(Validity) If an honest replica outputs v, v is an externally valid value, i.e., v is signed by a client.

(Termination) All honest replicas start with externally valid values, and all messages sent among them have been delivered, then honest replicas will output a value.

Forensic support. Traditionally, consensus protocols provide guarantees only when $f \leq t$. There can be a safety or liveness violation when $f > t$, which is the setting of study throughout this chapter. Our goal is to provide *forensic support* whenever there is a safety violation (or agreement violation) and the number of Byzantine replicas in the system is not too large. In particular, if the actual number of Byzantine faults is bounded by m (for some $m > t$) and there is a safety violation, we can detect d Byzantine replicas using a *forensic protocol*. The protocol takes as input the transcripts of honest parties and outputs the *irrefutable* proof of d culprits. With the irrefutable proof, any party (not necessarily in the BFT system) can be convinced of the culprits' identities *without* any assumption on the number of honest replicas. However, if even with transcripts from all honest replicas, no forensic protocol can output such a proof, the consensus protocol has no forensic support (denoted as "None" in Table 4.2). Note that when we say a protocol has no forensic support, we are referring to an impossibility w.r.t. providing irrefutable proof for d culprits (more precisely, in the context of Definition 4.3). In a general sense, there are other ways to provide forensics related to hardware, software, and network except for non-repudiation in the protocol.

To provide forensic support, we consider a setting where a client observes the existence of outputs for two conflicting (unequal) values.[1] By running a forensic protocol, the client sends (possibly a subset of) these conflicting outputs to all replicas and waits for their replies. Some of these replicas may be "witnesses" and may have (partial) information required to construct the irrefutable proof. After receiving responses from the replicas, the client constructs the proof. We denote by k the total number of transcripts from different honest replicas that are stored by the client to construct the proof.

Definition 4.3. $(\mathbf{m}, \mathbf{k}, \mathbf{d})$**-Forensic Support**. If $t < f \leq m$ and two honest replicas output conflicting values, then using the transcripts of all messages received from k honest replicas during the protocol, a client can provide an irrefutable proof of culpability of at least d Byzantine replicas.

[1]We assume all the (honest) replica outputs are eventually learned by the client. In practice, the client may monitor the outputs by periodically communicating with all replicas.

Other assumptions. All the protocols we consider in this chapter have their own network assumptions. For PBFT and HotStuff, we assume a partially synchronous network [5]. For VABA [3] and Algorand [9], we suppose asynchronous and synchronous networks respectively.

We assume all messages are digitally signed except for one variant of PBFT (§4.4.3) that sometimes relies on the use of Message Authenticated Codes (MACs). Some protocols, e.g., HotStuff, VABA, use threshold signatures. For forensic purposes, we assume multi-signatures [102] instead (possibly worsening the communication complexity in the process). Whenever the number of signatures exceeds a threshold, the resulting aggregate signature is denoted by a pair $\sigma = (\sigma_{agg}, \epsilon)$, where ϵ is a bitmap indicating whose signatures are included in σ_{agg}. We define the intersection of two aggregated messages to be the set of replicas who sign both messages, i.e., $\sigma \cap \sigma' := \{i | \sigma.\epsilon[i] \wedge \sigma'.\epsilon[i] = 1\}$. An aggregate signature serves as a quorum certificate (QC) in our protocols, and we will use the two terms interchangeably. We assume a collision resistant cryptographic hash function.

4.4 FORENSIC SUPPORT FOR PBFT

PBFT is a classical partially synchronous consensus protocol that provides an optimal resilience of t Byzantine faults out of $n = 3t + 1$ replicas. However, when the actual number of faults f exceeds t, it does not provide any safety or liveness. In this section, we show that when $f > t$ and in case of a safety violation, the variant of the PBFT protocol (referred to as PBFT-PK) where all messages sent by parties are signed, has the *strongest forensic support*. Further, we show that for an alternative variant where parties sometimes only use MACs (referred to as PBFT-MAC), forensic support is impossible.

4.4.1 Overview

We start with an overview focusing on a single-shot version of PBFT, i.e., a protocol for consensus on a single value. The protocol described here uses digital signatures to authenticate all messages and routes messages through leaders as shown in [103]; however we note that our arguments for PBFT-PK also apply to the original protocol in [7].

The protocol proceeds in a sequence of consecutive views denoted as view number $e = 1, 2, \cdots$. A higher view is a view with a larger view number. Each view has a unique leader. Each view of PBFT progresses as follows:

- **Pre-prepare.** The leader proposes a NewView message containing a proposal v and a status certificate M (explained later) to all replicas.

- **Prepare.** On receiving the first NewView message containing a valid value v in a view e, a replica sends Prepare for v if it is *safe* to vote based on a locking mechanism (explained later). It sends this vote to the leader. The leader collects $2t+1$ such votes to form an aggregate signature *prepareQC*. The leader sends *prepareQC* to all replicas.
- **Commit.** On receiving a *prepareQC* in view e containing message v, a replica locks on (v, e) and sends Commit to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *commitQC*. The leader sends *commitQC* to all replicas.
- **Reply.** On receiving *commitQC* from the leader, replicas output v and send a Reply (along with *commitQC*) to the client.

Once a replica locks on a value v in view e, we call (v, e) is the current lock of this replica. And a higher lock is a lock formed in a higher view. With lock (v, e), the replica only votes for the value v in subsequent views. The only scenario in which it votes for a value $v' \neq v$ is when the status certificate M provides sufficient information stating that $2t + 1$ replicas are not locked on v. At the end of a view, every replica sends its lock to the leader of the next view. The next view leader collects $2t + 1$ such values as a status certificate M.

The safety of PBFT comes from two key quorum intersection arguments:

Uniqueness within a view. Within a view, safety is ensured by votes in either round. Since a replica only votes once for the first valid value it receives, by a quorum intersection argument, two conflicting values cannot both obtain *commitQC* when $f \leq t$.

Safety across views. Safety across views is ensured by the use of locks and the status certificate. First, observe that if a replica r outputs a value v in view e, then a quorum of replicas lock on (v, e). When $f \leq t$, this quorum includes a set H of at least $t + 1$ honest replicas. For any replica in H to update to a higher lock, they need a *prepareQC* in a higher view $e' > e$, which in turn requires a vote from at least one of these honest replicas in view e'. However, replicas in H will vote for a conflicting value v' in a higher view only if it is accompanied by a status certificate M containing $2t+1$ locks that are not on value v. When $f \leq t$, the intersection of honest replicas in M with those in H has at least one replica – this honest replica will not vote for a conflicting value v'.

For completeness, we also provide a complete description of the PBFT-PK protocol in Algorithm 4.1. In the protocols in this chapter, we assume that replicas and clients ignore messages with invalid signatures and messages containing external invalid values. When searching for an entity (e.g., lock or *prepareQC*) with the highest view, break ties by alphabetic order of the value. Notice that ties only occur when $f > t$ and Byzantine replicas deliberately construct conflicting quorum certificates in a view.

Algorithm 4.1 PBFT-PK protocol: replica's initial value v_i

$LOCK \leftarrow (0, v_\perp, \sigma_\perp)$ with selectors e, v, σ ▷ $0, v_\perp, \sigma_\perp$: default view, value, and signature
$e \leftarrow 1$
while true **do**
▷ **Pre-prepare and Prepare Phase**
  as a leader
    collect $\langle \text{VIEWCHANGE}, e-1, \cdot \rangle$ from $2t+1$ distinct replicas as status certificate M ▷ Assume special VIEWCHANGE messages from view 0
    $v \leftarrow$ the locked value with the highest view number in M
    if $v = v_\perp$ **then**
      $v \leftarrow v_i$
    broadcast $\langle \text{NEWVIEW}, e, v, M \rangle$
  as a replica
    wait for valid $\langle \text{NEWVIEW}, e, v, M \rangle$ from leader ▷ Use function VALID($\langle \text{NEWVIEW}, e, v, M \rangle$)
    send $\langle \text{PREPARE}, e, v \rangle$ to leader
▷ **Commit Phase**
  as a leader
    collect $\langle \text{PREPARE}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow$ *aggregate-sign*(Σ)
    broadcast $\langle \text{COMMIT}, e, v, \sigma \rangle$
  as a replica
    wait for $\langle \text{COMMIT}, e, v, \sigma \rangle$ from leader ▷ *prepareQC*
    $LOCK \leftarrow (e, v, \sigma)$
    send $\langle \text{COMMIT}, e, v \rangle$ to leader

4.4.2 Forensic Analysis for PBFT-PK

The agreement property for PBFT holds only when $f \leq t$. When the number of faults is larger, this agreement property (and even termination) can be violated. In this section, we show how to provide forensic support for PBFT when the agreement property is violated. We show that if two honest replicas output conflicting values v and v' due to the presence of $t < f \leq m$ Byzantine replicas, our forensic protocol can detect $t+1$ Byzantine replicas with an irrefutable proof. For each of the possible scenarios in which safety can be violated, the proof shows exactly what property of PBFT was not respected by the Byzantine replicas. The irrefutable proof explicitly uses messages signed by the Byzantine parties, and is thus only applicable to the variant PBFT-PK where all messages are signed.

Intuition. In order to build intuition, let us assume $n = 3t+1$ and $f = t+1$ and start with a simple scenario: two honest replicas output values v and v' in the same view. It

Algorithm 4.1 (cont.)

▷ **Reply Phase**

as a leader
  collect $\langle \text{COMMIT}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
  $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
  broadcast $\langle \text{REPLY}, e, v, \sigma \rangle$
as a replica
  wait for $\langle \text{REPLY}, e, v, \sigma \rangle$ from leader ▷ *commitQC*
  output v and send $\langle \text{REPLY}, e, v, \sigma \rangle$ to the client
call procedure VIEWCHANGE()
if a replica encounters timeout in any "wait for", call procedure VIEWCHANGE()
procedure VIEWCHANGE()
  broadcast $\langle \text{BLAME}, e \rangle$
  collect $\langle \text{BLAME}, e \rangle$ from $t+1$ distinct replicas, broadcast them
  quit this view
  send $\langle \text{VIEWCHANGE}, e, LOCK \rangle$ to the next leader
  enter the next view, $e \leftarrow e+1$
function VALID($\langle \text{NEWVIEW}, e, v, M \rangle$)
  $v^* \leftarrow$ the locked value with the highest view number in M
  if $(v^* = v \vee v^* = v_\perp) \wedge (M$ contains locks from $2t+1$ distinct replicas$)$ **then**
    return $true$
  else
    return $false$

must then be the case that a *commitQC* is formed for both v and v'. Due to a quorum intersection argument, it must be the case that all replicas in the intersection have voted for two conflicting values to break the uniqueness property. Thus, all the replicas in the intersection are culpable. For PBFT-PK, the *commitQC* (as well as *prepareQC*) for the two conflicting values act as the irrefutable proof for detecting $t+1$ Byzantine replicas.

When two honest replicas output conflicting values in different views, there are many different sequences of events that could lead to such a disagreement. One such sequence is described in Figure 4.1. The replicas are split into three sets: the blue, the green, and the red set. The blue set and the green set are honest replicas with size t, while the red set is Byzantine replicas with size $t+1$.

- In view e, replica i outputs v due to *commitQC* formed with the COMMIT from the honest blue set and the Byzantine red set. At the end of the view, replicas in the blue and red set hold locks (v, e) whereas the green set holds a lower lock for a different value.
- In the next few views, no higher locks are formed. Thus, the blue and the red set still hold locks (v, e).

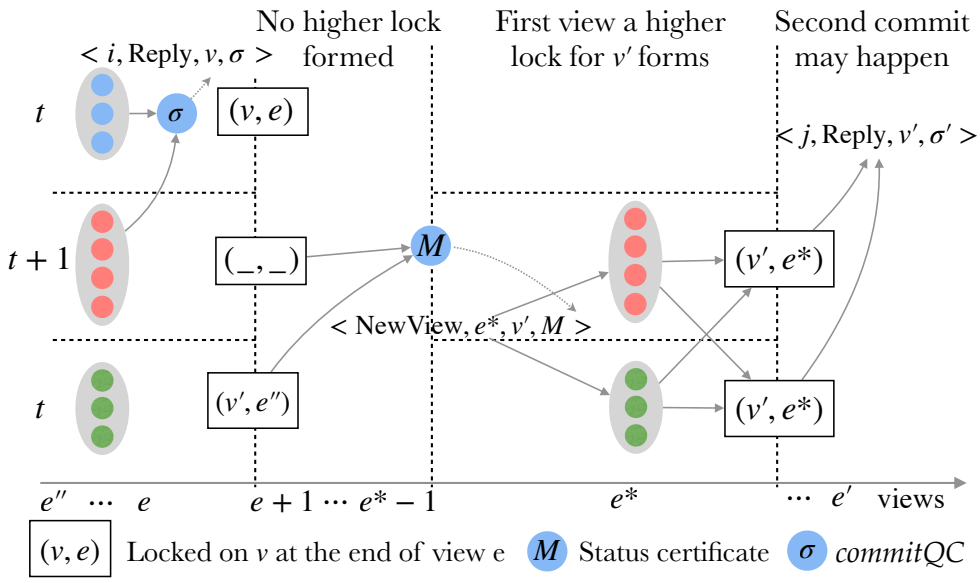


Figure 4.1: **An example sequence of events in the PBFT-PK protocol that leads to replicas i and j outputting different values.**

- Suppose e^* is the first view where a higher lock is formed. At the start of this view, the leader receives locks from the honest green set who holds lower-ranked locks and the Byzantine red set who maliciously sends lower-ranked locks. The set of locks received by the leader is denoted by M. Suppose the highest lock is received for v'. The leader proposes v' along with M. This can make any honest replica "unlock" and vote for v' and form quorum certificates in this view.
- In some later view e', replica j outputs v'.

With this sequence of events, consider the following questions: (1) Is this an admissible sequence of events? (2) How do we find the culpable Byzantine replicas? What does the irrefutable proof consist of? (3) How many replica transcripts do we need to construct the proof?

To answer the first question, the only nontrivial part in our example is the existence of a view e^* where a higher lock is formed. However, such a view $e < e^* \leq e'$ must exist because replica j outputs v' in view e' and a higher lock must be formed no later than view e'.

For the second question, observe that both the red replicas as well as the green replicas sent locks lower than (v, e) to the leader in e^*. However, only the red replicas also sent Commit messages for value v in view e. Thus, by intersecting the set of Commit messages for value v in view e and the messages forming the status certificate sent to the leader of e^*, we can obtain a culpable set of $t+1$ Byzantine replicas. So the proof for PBFT-PK consists of the *commitQC* in e and the status certificate in e^*, which indicates that the replicas sent a lower lock in view e^* despite having access to a higher lock in a lower view e.

For the third question, the NewView message containing the status certificate M in view e^* can act as the proof, so only one transcript needs to be stored.

Algorithm 4.2 Forensic protocol for PBFT-PK Byzantine agreement

```
1: as a replica running PBFT-PK
2:     Q ← all NewView messages in transcript
3:     upon receiving ⟨Request-Proof, e, v, e'⟩ from a client do
4:         for m ∈ Q do
5:             (v'', e'') ← the highest lock in m.M
6:             if (m.e ∈ (e, e']) ∧ (v'' ≠ v) ∧ (e'' ≤ e) then
7:                 send ⟨NewView, m⟩ to the client
8: as a client
9:     upon receiving two conflicting Reply messages do
10:        if the two messages are from different views then
11:            ⟨Reply, e, v, σ⟩ ← the message from lower view
12:            e' ← the view number of Reply from higher view
13:            broadcast ⟨Request-Proof, e, v, e'⟩
14:            wait for: ⟨NewView, m⟩ s.t. m.e ∈ (e, e'] ∧ (v'' ≠ v) ∧ (e'' ≤ e) where (v'', e'')
    is the highest lock in m.M.
15:            if in m.M there are two locks (e'', v_1, σ_1) and (e'', v_2, σ_2) s.t. v_1 ≠ v_2 then
16:                output σ_1 ∩ σ_2
17:            else
18:                output the intersection of senders in m.M and signers of σ.
19:        else
20:            ⟨Reply, e, v, σ⟩ ← first Reply message
21:            ⟨Reply, e, v', σ'⟩ ← second Reply message
22:            output σ ∩ σ'
```

Forensic protocol for PBFT-PK. Algorithm 4.2 describes the entire protocol to obtain forensic support atop PBFT-PK. Each replica keeps all received messages as transcripts and maintains a set Q containing all received NewView messages (line 2). If a client observes the replies of two conflicting values, it first checks if two values are output in the same view (line 9). If yes, then any two *commitQC* for two different output values can provide a culpability proof for at least $t+1$ replicas (lines 19-22). Otherwise, the client sends a request for a possible proof between two output views e, e' (lines 13). Each replica looks through the set Q for the NewView message in the smallest view $e^* > e$ such that the status certificate M contains the highest lock (e'', v'') where $v'' \neq v$ and $e'' \leq e$ and sends it to the client (line 7). If inside M there are conflicting locks in the same view, the intersection of them proves at least $t + 1$ culprits (line 15), otherwise the intersection of M and the *commitQC* proves at least $t + 1$ culprits (line 18).

The following theorem sharply characterizes the forensic support capability of PBFT-PK. As long as $m \leq 2t$, the best possible forensic support is achieved (i.e., $k = 1$ and $d = t + 1$). Algorithm 4.2 can be used to irrefutably detect $t+1$ Byzantine replicas. Conversely, if $m > 2t$ then no forensic support is possible (i.e., $k = n - f$ (messages from all honest nodes) and $d = 0$).

Theorem 4.1. With $n = 3t+1$, when $f > t$, if two honest replicas output conflicting values, PBFT-PK provides $(2t,\ 1,\ t+1)$-forensic support. Further $(2t+1, n-f, d)$-forensic support is impossible with $d > 0$.

Proof. We firstly prove the forward part of the theorem below. Suppose the values v and v' are output in views e and e' respectively.

Case $e = e'$. Culpability. The quorums *commitQC* for v and *commitQC* for v' intersect in $t + 1$ replicas. These $t + 1$ replicas should be Byzantine since the protocol requires a replica to vote for at most one value in a view.
Witnesses. Client can obtain the culpability proof based on two *commitQC*. No additional communication is needed in this case ($k = 0$).

Case $e \neq e'$. Culpability. If $e \neq e'$, then WLOG, suppose $e < e'$. Since v is output in view e, it must be the case that $2t + 1$ replicas are locked on (v, e) at the end of view e (if they are honest). Now consider the first view $e < e^* \leq e'$ in which a higher lock (v'', e^*) is formed (not necessarily known to any honest party) where $v'' \neq v$ (possibly $v'' = v'$). Such a view must exist since v' is output in view $e' > e$ and a lock will be formed in at least view e'. Consider the status certificate M sent by the leader of view e^* in its NewView message. M must contain $2t + 1$ locks; each of these locks must be from view $e'' \leq e$, and a highest lock among them is (v'', e'').

We consider two cases based on whether the status certificate contains two different highest locks: (i) there exist two locks (v'', e'') and (v''', e'') s.t. $v'' \neq v'''$ in M. (ii) (v'', e'') is the only highest lock in M. For the first case, since two locks are formed in the same view, the two quorums forming the two locks in view e'' intersect in $t + 1$ replicas. These replicas are Byzantine since they voted for more than one value in view e.

For the second case, (v'', e'') is the only highest lock in the status certificate M. M intersects with the $2t + 1$ signers of *commitQC* in view e at $t + 1$ Byzantine replicas. These replicas are Byzantine because they had a lock on $v \neq v''$ in view $e \geq e''$ but sent a different lock to the leader of view $e^* > e$.

Witnesses. Client can obtain the proof by storing the NewView message containing the status certificate M in e^*. Only one witness is needed to provide the NewView message ($k = 1$). The status certificate M and the first *commitQC* act as the irrefutable proof.

Then we prove the converse (impossibility) part. Suppose there are $f = 2t + 1$ Byzantine replicas, and let there be three replica partitions P, Q, R, $|P| = |Q| = t$, $|R| = t + 1$. To prove the result, suppose the protocol has forensic support for $d > 0$, we construct two worlds where a different set of replicas is Byzantine in each world.

World 1. Let R, Q be Byzantine replicas in this world. During the protocol, replicas in Q behave like honest parties. Suppose in view e, e' ($e < e'$), two honest replicas $p_1, p_2 \in P$ output two conflicting values v, v' after receiving two *commitQC*. The *commitQC* for v contains the Commit messages from P and R, and the *commitQC* for v' contains the Commit messages from R and Q. All the other messages never reach P. During the forensic protocol, replicas in P send their transcripts to the client. Since the protocol has forensic support for $d > 0$, the forensic protocol determines that some replicas in R are culpable (since Q behave like honest), using these transcripts (two *commitQC*).

World 2. Let P, Q and some replica $r \in R$ are Byzantine replicas and replicas in r behave honestly. Again, in view e, e' ($e < e'$), two replicas $p_1, p_2 \in P$ output two conflicting values v, v' after receiving two *commitQC*. The *commitQC* for v contains the Commit messages from P and R, and the *commitQC* for v' contains the Commit messages from R and Q. Replicas in R unlock themselves due to receiving a higher *prepareQC* formed in e^* ($e < e^* < e'$). During the forensic protocol, replicas in P send the same transcripts as of World 1 to the client (only two *commitQC*). Thus, the forensic protocol outputs some replicas in R as culpable ones. However, this is incorrect since replicas in R are honest (r is indistinguishable from replicas in $R/\{r\}$).

QED.

Communication complexity. In the first branch of the forensic protocol, Algorithm 4.2, the client needs to receive one message from $k = 1$ replica and the message size is $(2t + 1)(|v| + |sig|)$ where $|v|$ and $|sig|$ stand for the size of a value and an aggregate signature (line 14). In the second branch, the client doesn't need any message (line 19). Therefore the complexity of the client receiving messages is $O(n(|v| + |sig|))$. Notice that we exclude the communication for learning replica outputs (Reply messages) since that procedure happens *before* the forensic protocol.

4.4.3 Impossibility for PBFT-MAC

We now show an impossibility for a variant of PBFT proposed in [7, Section 5]. The arguments here also apply to the variant in [8]. Compared to §4.4.1, the only difference in this variant is (i) PREPARE and COMMIT messages are authenticated using MACs instead of signatures, and (ii) these messages are broadcast instead of routing them through the leader.

Intuition. The key intuition behind the impossibility relies on the absence of digital signatures which were used to "log" the state of a replica when some replica i outputs a value. In particular, if we consider the example in Figure 4.1, while i receives $2t+1$ COMMIT messages for value v, these messages are not signed. Thus, if $t+1$ Byzantine replicas vote for a different value v', v' can be output by a different replica. The absence of a verifiable proof stating the set of replicas that sent a COMMIT to replica i prevents any forensic analysis. We formalize this intuition below.

Theorem 4.2. With $n = 3t+1$, when $f > t$, if two honest replicas output conflicting values, $(t+1, 2t, d)$-forensic support is impossible with $d > 0$ for PBFT-MAC.

Proof. Suppose the protocol provides forensic support to detect $d \geq 1$ replicas with irrefutable proof. To prove this result, we construct two worlds where a different set of $t+1$ replicas is Byzantine in each world but a forensic protocol cannot be correct in both worlds. We fix $f = t+1$, although the arguments will apply for any $f > t$.

Let there be four replica partitions $P, Q, R, \{x\}$. $|P| = |Q| = |R| = t$, and x is an individual replica. In both worlds, the conflicting outputs are presented in the same view e. Suppose the leader is a replica from set R.

World 1. Let P and x be Byzantine replicas in this world. The honest leader from set R in view e proposes v'. Parties in R, x and Q send PREPARE and COMMIT messages (authenticated with MACs) for value v'. Due to partial synchrony, none of these messages arrive at P. At the end of view e, only R and one replica q in Q receive enough COMMIT messages and send replies to the client. So the client receives the first set of $t+1$ replies for value v', which contains the same quorum R, x, Q.

The Byzantine parties in P and x simulate a proposal from the leader for v and the sending of PREPARE and COMMIT messages within R, P and x. The simulation is possible due to the absence of a PKI. At the end of view e, P and x obtain enough COMMIT messages and send replies to the client. Thus, the client receives the second set of $t+1$ replies for value v, which contains the same quorum P, R, x. Client starts the forensic protocol.

During the forensic protocol, Byzantine P and x present their votes for v and they also forge votes from R for v as their transcripts. As $t+1$ parties have output each of v and v', there is a safety violation. Since the protocol has forensic support for $d \geq 1$, using these transcripts, the forensic protocol determines some replicas in P and x are culpable.

World 2. Let R and q (one replica in Q) be Byzantine replicas in this world. The Byzantine leader in view e proposes v to P, R and x. They send PREPARE and COMMIT messages (authenticated with MACs) for value v. These messages do not arrive at Q. At the end of view e, parties in P and x output v. So the client receives the first set of $t+1$ replies for value v, which contains the same quorum P, R, x.

Similarly, the leader sends v' to Q, R and x. The proposal does not arrive at x. Only Q and R send PREPARE and COMMIT messages (authenticated with MACs) for v', these messages do not arrive at P. However, R and q forge PREPARE and COMMIT messages from x. At the end of view e, R and q output v'. So the client receives the second set of $t+1$ replies for value v', which contains the same quorum R, x, Q. Client starts the forensic protocol.

During the forensic protocol, Byzantine R and q send the same transcripts as in World 1 by dropping votes for v and forging votes from x. Again, since $t+1$ parties have output each of v and v', there is a safety violation. However, observe that the transcript presented to the forensic protocol is identical to that in World 1. Thus, the forensic protocol outputs some replicas in P and x as culpable replicas. In World 2, this is incorrect since replicas in P and x are honest.

QED.

4.5 FORENSIC SUPPORT FOR HOTSTUFF

HotStuff [2] is a partially synchronous consensus protocol that provides an optimal resiliency of $n = 3t + 1$. The HotStuff protocol is similar to PBFT but there are subtle differences which allow it to obtain a linear communication complexity for both its steady state and view change protocols (assuming the presence of threshold signatures). Looking ahead, these differences significantly change the way forensics is conducted if a safety violation happens.

4.5.1 Overview

We start with an overview of the protocol. For simplicity, we discuss a single-shot version of HotStuff. The protocol proceeds in a sequence of consecutive views where each view has

a unique leader. Each view of HotStuff progresses as follows:[2]

- **Pre-prepare.** The leader proposes a NewView message containing a proposal v along with the *highQC* (the highest *prepareQC* known to it) and sends it to all replicas.
- **Prepare.** On receiving a NewView message containing a valid value v in a view e and a *highQC*, a replica sends Prepare for v if it is *safe* to vote based on a locking mechanism (explained later). It sends this vote to the leader. The leader collects $2t+1$ votes to form an aggregate signature *prepareQC* in view e. The leader sends the view e *prepareQC* to all replicas.
- **Precommit.** On receiving a *prepareQC* in view e containing message v, a replica updates its highest *prepareQC* to (v, e) and sends Precommit to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *precommitQC*.
- **Commit.** On receiving *precommitQC* in view e containing message v from the leader, a replica locks on (v, e) and sends Commit to the leader. The leader collects $2t + 1$ such votes to form an aggregate signature *commitQC*.
- **Reply.** On receiving *commitQC* from the leader, replicas output the value v and send a Reply (along with *commitQC*) to the client.

Once a replica locks on a given value v, it only votes for the value v in subsequent views. The only scenario in which it votes for a value $v' \neq v$ is when it observes a *highQC* from a higher view in a NewView message. At the end of a view, every replica sends its highest *prepareQC* to the leader of the next view. The next view leader collects $2t + 1$ such values and picks the highest *prepareQC* as *highQC*. The safety and liveness of HotStuff when $f \leq t$ comes from the following:

Uniqueness within a view. Since replicas only vote once in each round, a *commitQC* can be formed for only one value when $f \leq t$.

Safety and liveness across views. Safety across views is ensured using locks and the voting rule for a NewView message. Whenever a replica outputs a value, at least $2t + 1$ other replicas are locked on the value in the view. Observe that compared to PBFT, there is no status certificate M in the NewView message to "unlock" a replica. Thus, a replica only votes for the value it is locked on. The only scenario in which it votes for a conflicting value v' is when the leader includes a *prepareQC* for v' from a higher view in NewView message. This indicates that at least $2t+1$ replicas are not locked on v in a higher view, and hence it should be safe to vote for it. The latter constraint of voting for v' is not necessary

[2]The description of HotStuff protocol is slightly different from the basic algorithm described in [2, Algorithm 2] to be consistent with the description of PBFT in §4.4.1.

for safety, but only for liveness of the protocol.

Variants of HotStuff. In this chapter, we study three variants of single-shot HotStuff, identical for the purposes of consensus but varied for forensic support. The distinction among them is only in the information carried in PREPARE message. For all three versions, the message contains the message type PREPARE, the current view number e and the proposed value v. In addition, PREPARE in HotStuff-view contains e_{qc}, the view number of the *highQC* in the NEWVIEW message. HotStuff-hash contains the hash of *highQC* (cf. Table 4.3). HotStuff-hash is equivalent to the basic algorithm described in [2, Algorithm 2]. HotStuff-null does not add additional information.

Table 4.3: Comparison of different variants of HotStuff, the PREPARE message is $\langle \text{PREPARE}, e, v, \mathit{Info} \rangle$

	HotStuff-view	HotStuff-hash	HotStuff-null
Info	e_{qc}	Hash(*highQC*)	$\emptyset$
m	$2t$	$2t$	$t+1$
k	1	$t+1$	$2t$
d	$t+1$	$t+1$	1

4.5.2 Forensic Analysis for HotStuff

If two conflicting values are output in the same view, Byzantine replicas can be detected by using *commitQC* and ideas similar to that in PBFT. However, when the conflicting outputs of replicas i and j are across views e and e' for $e < e'$, the same ideas do not hold anymore. To understand this, observe that the two key ingredients for proving the culpability of Byzantine replicas in PBFT were (i) a *commitQC* for the value output in a lower view (denoted by σ for replica i's reply in Figure 4.1) and (ii) a status certificate from the first view higher than e containing the locks from $2t+1$ replicas (denoted by M for view $e^* > e$ in Figure 4.1). In HotStuff, a *commitQC* still exists. However, for communication efficiency reasons, HotStuff does not include a status certificate M in its NEWVIEW message. The status certificate in PBFT provides us with the following:

- **Identifying a potential set of culpable replicas.** Depending on the contents of M and knowing σ, we could identify a set of Byzantine replicas.
- **Determining whether the view is the first view where a higher lock for a conflicting value is formed.** By inspecting all locks in M, we can easily determine this. Ensuring first view with a higher lock is important. Once a higher lock is formed,

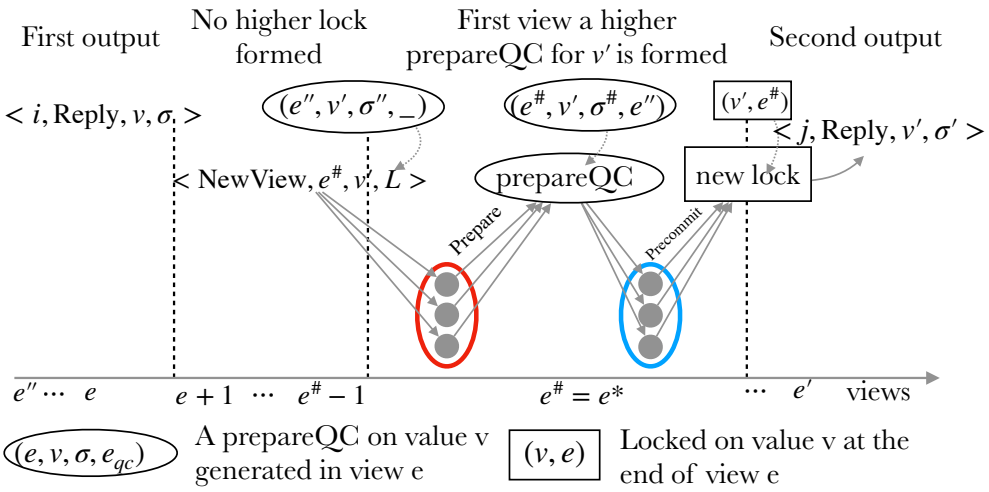


Figure 4.2: **Depiction of events in the HotStuff-view protocol for the first view where a higher *prepareQC* for v' is formed.**

even honest replicas may update their locks and the proof of Byzantine behavior may not exist in the messages in subsequent views.

Let us try to understand this based on the first view $e^{\#}$ where a higher *prepareQC* is formed for $v' \neq v$ (see Figure 4.2). The set of replicas which sent PREPARE (the red ellipse) in $e^{\#}$ and formed a *prepareQC* are our potential set of Byzantine replicas. Why? If $e^{\#}$ is indeed the first view in which a higher *prepareQC* is formed, then all of these replicas must have voted for a NEWVIEW message containing a *highQC* from a lower or equal view e'' on a different value. If any of these replicas also held a lock (v, e) (by voting for replica i's output), then these replicas must have output the culpable act of not respecting the voting rule.

The only remaining part is to ensure that this is indeed the first view where a higher conflicting *prepareQC* is formed. The way to prove this is also the key difference among three variants of HotStuff. For HotStuff-view, *prepareQC* contains e_{qc}, which explicitly states the view number of *highQC* in the NEWVIEW message they vote for. If $e_{qc} < e$, *prepareQC* provides an irrefutable proof for culpable behavior. For HotStuff-hash, the hash information contained in PREPARE provides the necessary link to the NEWVIEW message they vote, so once the linked NEWVIEW message is accessible, the *prepareQC* and NEWVIEW together serve as the proof for culpable behavior. However, for HotStuff-null, even if we receive both *prepareQC* and NEWVIEW messages that are formed in the same view, no proof can be provided to show a connection between them. A Byzantine node vote for the first higher *prepareQC* can always refuse to provide the NEWVIEW message they receive.

Thus, to summarize, the red set of replicas in view $e^{\#}$ is a potential set of culpable nodes of size $t + 1$. The irrefutable proof to hold them culpable constitutes two parts, (1) the

first *prepareQC* containing their signed PREPARE messages, and (2) a proof to show this is indeed the first view. In the next two subsections we will introduce the forensic protocols of HotStuff-view and HotStuff-hash and the impossibility of HotStuff-null to have neither forensic protocol nor forensic support.

For completeness, a description of the general HotStuff protocol is provided in Algorithm 4.3.

Algorithm 4.3 General HotStuff protocol: replica's initial value v_i, protocol variant indicator $var \in \{$'HotStuff-view', 'HotStuff-hash', 'HotStuff-null'$\}$

$prepareQC \leftarrow (0, v_\perp, \sigma_\perp, Info_\perp)$ with selectors $e, v, \sigma, Info$
$LOCK \leftarrow (0, v_\perp)$ with selectors e, v ▷ $0, v_\perp, \sigma_\perp, Info_\perp$: default view, value, signature, and info
$e \leftarrow 1$
while true **do**
▷ **Pre-prepare and Prepare Phase**
  as a leader
    collect $\langle \text{VIEWCHANGE}, e-1, \cdot \rangle$ from $2t+1$ distinct replicas as M ▷ Assume special VIEWCHANGE messages from view 0
    $highQC \leftarrow$ the highest QC in M
    $v \leftarrow highQC.v$
    if $v = v_\perp$ **then**
      $v \leftarrow v_i$
    broadcast $\langle \text{NEWVIEW}, e, v, highQC \rangle$
  as a replica
    wait for $\langle \text{NEWVIEW}, e, v, highQC \rangle$ from leader s.t. $highQC.v = v \vee highQC.v = v_\perp$ ▷ Validate v
    if $(LOCK.e < highQC.e) \vee (LOCK.v = v \wedge LOCK.e = highQC.e)$ **then**
      send $\langle \text{PREPARE}, e, v, \text{INFO}(var, highQC) \rangle$ to leader ▷ Use function $\text{INFO}(var, highQC)$
▷ **Precommit Phase**
  as a leader
    collect $\langle \text{PREPARE}, e, v, Info \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
    $\sigma \leftarrow aggregate\text{-}sign(\Sigma)$
    broadcast $\langle \text{PRECOMMIT}, e, v, \sigma, Info \rangle$
  as a replica
    wait for $\langle \text{PRECOMMIT}, e, v, \sigma, Info \rangle$ from leader ▷ *prepareQC*
    $prepareQC \leftarrow (e, v, \sigma, Info)$
    send $\langle \text{PRECOMMIT}, e, v \rangle$ to leader

Algorithm 4.3 (cont.)

▷ **Commit Phase**

as a leader
  collect $\langle \textsc{Precommit}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
  $\sigma \leftarrow \mathit{aggregate\text{-}sign}(\Sigma)$
  broadcast $\langle \textsc{Commit}, e, v, \sigma \rangle$
as a replica
  wait for $\langle \textsc{Commit}, e, v, \sigma \rangle$ from leader ▷ *precommitQC*
  $LOCK \leftarrow (e, v)$
  send $\langle \textsc{Commit}, e, v \rangle$ to leader

▷ **Reply Phase**

as a leader
  collect $\langle \textsc{Commit}, e, v \rangle$ from $2t+1$ distinct replicas, denote the collection as Σ
  $\sigma \leftarrow \mathit{aggregate\text{-}sign}(\Sigma)$
  broadcast $\langle \textsc{Reply}, e, v, \sigma \rangle$
as a replica
  wait for $\langle \textsc{Reply}, e, v, \sigma \rangle$ from leader ▷ *commitQC*
  output v and send $\langle \textsc{Reply}, e, v, \sigma \rangle$ to the client
call procedure ViewChange()

if a replica encounters timeout in any "wait for", call procedure ViewChange()
procedure ViewChange()
  broadcast $\langle \textsc{Blame}, e \rangle$
  collect $\langle \textsc{Blame}, e \rangle$ from $t+1$ distinct replicas, broadcast them
  quit this view
  send $\langle \textsc{ViewChange}, e, \mathit{prepareQC} \rangle$ to the next leader
  enter the next view, $e \leftarrow e+1$
function Info(var, highQC) ▷ $var \in$ {'HotStuff-view', 'HotStuff-hash', 'HotStuff-null'}
  if $var =$'HotStuff-view' **then**
    return $\mathit{highQC}.e$
  if $var =$'HotStuff-hash' **then**
    return Hash(highQC)
  if $var =$'HotStuff-null' **then**
    return $\emptyset$

4.5.3 Forensic Protocols for HotStuff-view and HotStuff-hash

Algorithm 4.4 Forensic protocol for HotStuff-view

1: **as** a replica running HotStuff-view
2: $P \leftarrow$ all *prepareQC* in transcript ▷ including *prepareQC* in PRECOMMIT message and *highQC* in NEWVIEW message
3: **upon receiving** $\langle$REQUEST-PROOF$, e, v, e'\rangle$ from a client **do**
4: **for** $qc \in P$ **do**
5: **if** $(qc.v \neq v) \wedge (qc.e \in (e, e']) \wedge (qc.e_{qc} \leq e)$ **then**
6: send $\langle$PROOF-ACROSS-VIEW$, qc\rangle$ to the client
7: **as** a client
8: **upon receiving** two conflicting REPLY messages **do**
9: **if** two messages are from different views **then**
10: $\langle$REPLY$, e, v, \sigma\rangle \leftarrow$ the message from lower view
11: $e' \leftarrow$ the view number of message from higher view
12: broadcast $\langle$REQUEST-PROOF$, e, v, e'\rangle$
13: wait for $\langle$PROOF-ACROSS-VIEW$, qc\rangle$ s.t.
(1) $e < qc.e \leq e'$, and
(2) $(qc.v \neq v) \wedge (qc.e_{qc} \leq e)$
14: **output** $qc.\sigma \cap \sigma$
15: **else**
16: $\langle$REPLY$, e, v, \sigma\rangle \leftarrow$ first REPLY message
17: $\langle$REPLY$, e, v', \sigma'\rangle \leftarrow$ second REPLY message
18: **output** $\sigma \cap \sigma'$

Forensic protocol for HotStuff-view. Algorithm 4.4 describes the protocol to obtain forensic support atop HotStuff-view. Each replica keeps all received messages as transcript and maintains a set P containing all received *prepareQC* from PRECOMMIT messages and *highQC* from NEWVIEW messages (line 2). If a client observes outputs of two conflicting values in the same view, it can determine the culprits using the two REPLY messages (line 15). Otherwise, the client sends a request to all replicas for a possible proof between two output views e, e' for $e < e'$ (line 12). Each replica looks through the set P for *prepareQC* formed in views $e < e^{\#} \leq e'$. If there exists a *prepareQC* whose value is different from the value v output in e and whose e_{qc} is less than or equal to e, it sends a reply with this *prepareQC* to the client (line 6). The client waits for a *prepareQC* (line 13) formed between two output views. For HotStuff-view, if it contains a different value from the first output value and an older view number $e_{qc} < e$, the intersection of this *prepareQC* and the *commitQC* from the REPLY message in the lower view proves at least $t + 1$ culprits (line 14).

Algorithm 4.5 Forensic protocol for HotStuff-hash

```
1: as a replica running HotStuff-hash
2:     P ← all prepareQC in transcript
3:     Q ← all NewView messages in transcript
4:     upon receiving ⟨Request-Proof, e, v, e'⟩ from a client do
5:         for qc ∈ P do
6:             if (qc.v ≠ v) ∧ (qc.e ∈ (e, e']) then
7:                 send ⟨Proof-across-View, qc⟩ to the client
8:         for m ∈ Q do
9:             if (m.v ≠ v) ∧ (m.e ∈ (e, e']) ∧ (m.highQC.e ≤ e) then
10:                send ⟨NewView, m⟩ to the client
11: as a client
12:    NV ← {}
13:    upon receiving two conflicting Reply messages do
14:        if two messages are from different views then
15:            ⟨Reply, e, v, σ⟩ ← the message from lower view
16:            e' ← the view number of message from higher view
17:            broadcast ⟨Request-Proof, e, v, e'⟩
18:            upon receiving ⟨NewView, m⟩ do
19:                if (m.v ≠ v) ∧ (m.e ∈ (e, e']) ∧ (m.highQC.e ≤ e) then
20:                    NV ← NV ∪ {m}
21:            wait for ⟨Proof-across-View, qc⟩ s.t.
               (1) e < qc.e ≤ e', and
               (2) (qc.v ≠ v) ∧ (∃m ∈ NV, Hash(m) = qc.hash)
22:            output qc.σ ∩ σ
23:        else
24:            ⟨Reply, e, v, σ⟩ ← first Reply message
25:            ⟨Reply, e, v', σ'⟩ ← second Reply message
26:            output σ ∩ σ'
```

Forensic protocol for HotStuff-hash. Algorithm 4.5 describes the protocol to obtain forensic support atop HotStuff-hash, which is similar to the protocol for HotStuff-view. For replicas running HotStuff-hash, besides P, they also maintain the set Q for received NewView messages (line 3). When receiving a forensic request from clients, replicas look through P for *prepareQC* formed in views $e < e^{\#} \leq e'$ and send all *prepareQC* whose values are different from the value v to the client (line 7). Besides, they also look through Q for a NewView message formed in views $e < e^{\#} \leq e'$ and send all NewView proposing a value different from v and containing a *highQC* with view number $\leq e$ (line 10). For HotStuff-hash, when receiving such a NewView for different values, the message will be stored temporarily by the client until a *prepareQC* for the NewView message with a matching hash is received. The NewView and the *prepareQC* together form the desired proof; the intersection of the

prepareQC and the *commitQC* provides at least $t + 1$ culprits.

Forensic proofs. The following theorems characterize the forensic support capability of HotStuff-view and HotStuff-hash. As long as $m \leq 2t$, HotStuff-view can achieve the best possible forensic support (i.e., $k = 1$ and $d = t + 1$). HotStuff-hash can achieve a medium forensic support (i.e., $k = t + 1$ and $d = t + 1$). Conversely, if $m > 2t$ then no forensic support is possible for both protocols (i.e., $k = n - f$ and $d = 0$).

Theorem 4.3. With $n = 3t+1$, when $f > t$, if two honest replicas output conflicting values, HotStuff-view provides $(2t,\ 1,\ t+1)$-forensic support. Further $(2t+1, n-f, d)$-forensic support is impossible with $d > 0$.

Proof. We prove the forward part of the theorem below. Suppose two conflicting values $v,\ v'$ are output in views $e,\ e'$ respectively.

Case $e = e'$. Culpability. The *commitQC* of v and *commitQC* of v' intersect in $t+1$ replicas. These $t + 1$ replicas should be Byzantine since the protocol requires a replica to vote for at most one value in a view.
Witnesses. Client can obtain a proof based on the two Reply messages, so additional witnesses are not necessary in this case.

Case $e \neq e'$. Culpability. If $e \neq e'$, then WLOG, suppose $e < e'$. Since v is output in view e, it must be the case that $2t + 1$ replicas are locked on (v, e) at the end of view e. Now consider the first view $e < e^* \leq e'$ in which a higher lock (v'', e^*) is formed where $v'' \neq v$ (possibly $v'' = v'$). Such a view must exist since v' is output in view $e' > e$ and a lock will be formed in at least view e'. For a lock to be formed, a higher *prepareQC* must be formed too. Consider the first view $e < e^{\#} \leq e'$ in which the corresponding *prepareQC* for v'' is formed. The leader in $e^{\#}$ broadcasts the NewView message containing a *highQC* on (v'', e''). Since this is the first time a higher *prepareQC* is formed and there is no higher *prepareQC* for v'' formed between view e and $e^{\#}$, we have $e'' \leq e$. The formation of the higher *prepareQC* indicates that $2t+1$ replicas have received the NewView message proposing v'' with *highQC* on (v'', e'') and consider it a valid proposal, i.e., the view number e'' is larger than their locks because the value is different.

Recall that the output value v indicates $2t + 1$ replicas are locked on (v, e) at the end of view e. In this case, the $2t + 1$ votes in *prepareQC* in view $e^{\#}$ intersect with the $2t + 1$ votes in *commitQC* in view e at $t+1$ Byzantine replicas. These replicas should be Byzantine because they have been locked on the value v in view e and voted for a value $v'' \neq v$ in a

higher view $e^{\#}$ when the NewView message contains a *highQC* from a view $e'' \leq e$. Thus, they have violated the voting rule.
Witnesses. Client can obtain a proof by storing a *prepareQC* formed between e and e', whose value is different from v and whose $e_{qc} \leq e$. So only one witness is needed ($k = 1$), and the *prepareQC* and the first *commitQC* act as the irrefutable proof.

The proof of converse (impossibility) is the same as Theorem 4.1. QED.

Theorem 4.4. With $n = 3t+1$, when $f > t$, if two honest replicas output conflicting values, HotStuff-hash provides $(2t,\ t+1,\ t+1)$-forensic support. Further $(2t+1, n-f, d)$-forensic support is impossible with $d > 0$.

Proof. We prove the forward part of the theorem below. Suppose two conflicting values $v,\ v'$ are output in views e, e' respectively.

Case $e = e'$. Same as Theorem 4.3.

Case $e \neq e'$. Culpability. Same as Theorem 4.3.
Witnesses. Since *prepareQC* of HotStuff-hash only has the hash of *highQC*, the irrefutable proof contains the NewView message that includes the *highQC* and the corresponding *prepareQC* with the matching hash Hash(*highQC*). The client may need to store all NewView messages between e and e' whose value is different from v and whose *highQC* is formed in $e_{qc} \leq e$, until receiving a *prepareQC* for some NewView message with a matching hash. In the best case, some replica sends both the NewView message and the corresponding *prepareQC*, so the client only needs to store $k = 1$ replica's transcript. In the worst case, we can prove that any $t + 1$ messages of transcript are enough to get the proof. Consider the honest replicas who receive the first *prepareQC* and the NewView message. $2t + 1$ replicas have access to the *prepareQC*, and the other $2t + 1$ replicas have access to the NewView message. Among them at least $t + 1$ replicas have access to both messages, and we assume they are all Byzantine. Then t honest replicas (or more) have the *prepareQC*, and the other t honest replicas (or more) have the NewView message. The total number of honest replicas $n - f \leq 2t$. Thus among any $t+1$ honest replicas, at least one has NewView message and at least one (probably the other one) has *prepareQC*. Therefore, $t+1$ transcripts from honest replicas ensure the access of both NewView message and *prepareQC* and thus guarantee the irrefutable proof.

The proof of converse (impossibility) is the same as Theorem 4.1. QED.

Communication complexity. In line 13 of Algorithm 4.4, the client needs to receive one message from $k = 1$ replica and the message size is $(|v| + |sig|)$ where $|v|$ and $|sig|$ stand

for the size of a value and an aggregate signature. Therefore the complexity of the client receiving messages is $O(|v| + |sig|)$ for HotStuff-view. As for HotStuff-hash, theorem 4.4 shows that in the worst case, the client needs to receive messages from $k = t + 1$ replicas. Each of those replicas sends one message of size $O(|v| + |sig| + |hash|)$ where $|hash|$ stands for the size of a hash value. Therefore the complexity of the client receiving messages is $O(n(|v| + |sig| + |hash|))$ for HotStuff-hash.

4.5.4 A Forensic Attack on HotStuff-view

Compared to HotStuff [2, Algorithm 2], Algorithm 4.3 has a slightly different voting rule in line 14. In addition to checking whether $(LOCK.e < highQC.e) \vee (LOCK.v = v)$ holds as in HotStuff [2, Algorithm 2], when the value in NewView is the same as the value in lock, our voting rule requires $LOCK.e = highQC.e$.

We argue that the lack of this additional check on the view number will not affect the safety and liveness for HotStuff, but pose a threat for forensics. In the following, we exhibit a forensic attack on HotStuff-view protocol with the original voting rule.

- $e = i > 0$: An honest replica R receives a $\langle \text{Commit}, i, v, \sigma \rangle$ from the leader and updates its lock to be (i, v, σ). R sends $\langle \text{Commit}, i, v \rangle$ to leader, which is contained in a *commitQC* denoted as qc_1. v is output in this view.
- $e = i + 1$: R receives $\langle \text{Commit}, i + 1, v', \sigma' \rangle$ and updates its lock to be $(i + 1, v', \sigma')$.
- $e = i + 2$: A leader broadcasts $\langle \text{NewView}, i + 2, v', highQC \rangle$, where *highQC* is a QC from $i - 1$. Replica R receives the message, checks the original voting rule, and sends $\langle \text{Prepare}, i + 2, v', i - 1 \rangle$, because $LOCK.v = v'$. This message is contained in a *prepareQC* denoted as qc_2. Further, v' is output in this view.

In this execution, replica R follows the protocol, however, it will be mistakenly blamed by Algorithm 4.4 if the client receives the qc_1 for v and the qc_2 for v'. Since $qc_2.v \neq qc_1.v$ and $qc_2.e_{qc} = i - 1 \leq qc_1.e = i$ according to line 10.

While the actual *prepareQC* whose intersection with qc_1 should be blamed is generated in $e = i + 1$, it is possible that some honest replicas who have the same transcripts as R will be improperly held culpable in this case. By adding the condition to check $LOCK.e = highQC.e$, honest replicas will not vote for a NewView with stale *highQC*, which prevents them from the attack described above.

4.5.5 Impossibility for HotStuff-null

In HotStuff-null, Prepare message and *prepareQC* are not linked to the NewView message. We show that this lack of information is sufficient to disable forensic support.

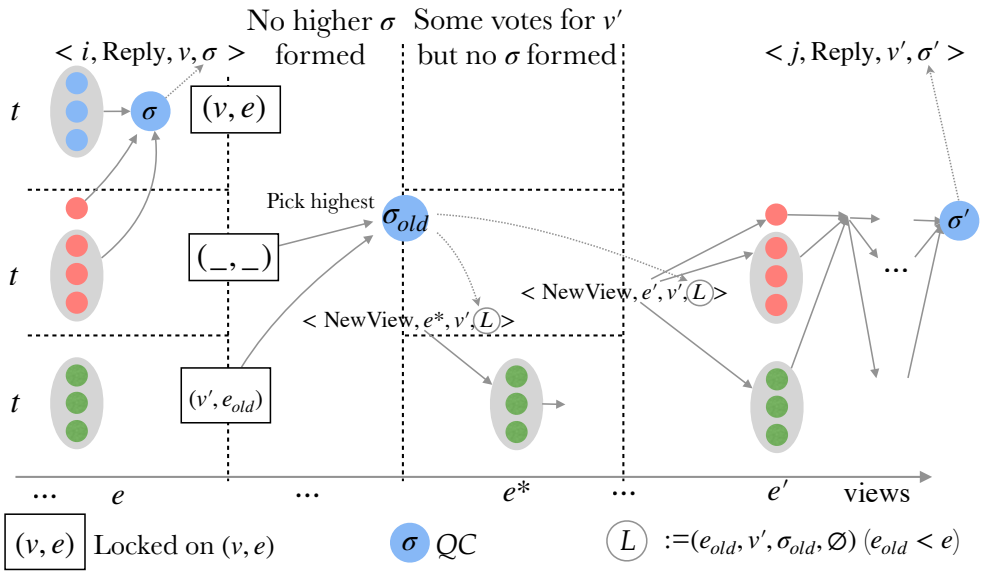


Figure 4.3: World 1 of Theorem 4.5. Replicas are represented as colored nodes. Replica partitions are P, $\{x\}$ (Byzantine), R (Byzantine), and Q from top to bottom.

Intuition. When $f = t+1$, from the forensic protocols of HotStuff-view and HotStuff-hash, we know that given across-view $commitQC_1$ and $commitQC_2$ (ordered by view) and the first $prepareQC$ higher than $commitQC_1$, the intersection of $prepareQC$ and $commitQC_1$ contains at least $d = t + 1$ Byzantine replicas. The intersection argument remains true for HotStuff-null, however, it is impossible for a client to decide whether $prepareQC$ is the first one only with the transcripts sent by $2t$ honest replicas (when $f = t+1$). In an execution where there are two $prepareQC$ in view e^* and e' respectively ($e^* < e'$), the Byzantine replicas (say, set P) may not respond with the $prepareQC$ in e^*. The lack of information disallows a client from separating this world from another world where P is indeed honest and shares all the information available. We formalize this intuition in the theorem below.

Theorem 4.5. With $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting values, $(t + 1,\ 2t,\ d)$-forensic support is impossible with $d > 1$ for HotStuff-null. Further, $(t + 2,\ n - f,\ d)$-forensic support is impossible with $d > 0$.

Proof. Suppose the protocol provides forensic support to detect $d > 1$ Byzantine replicas with irrefutable proof which can be constructed from the transcripts of all honest replicas. To prove this result, we construct two worlds where a different set of replicas is Byzantine in each world. We will fix the number of Byzantine replicas $f = t + 1$, but the following argument works for any $f \geq t + 1$.

Let there be four replica partitions $P, Q, R, \{x\}$. $|Q| = |P| = |R| = t$, and x is an individual replica. In both worlds, the conflicting outputs are presented in view e, e' $(e + 1 < e')$

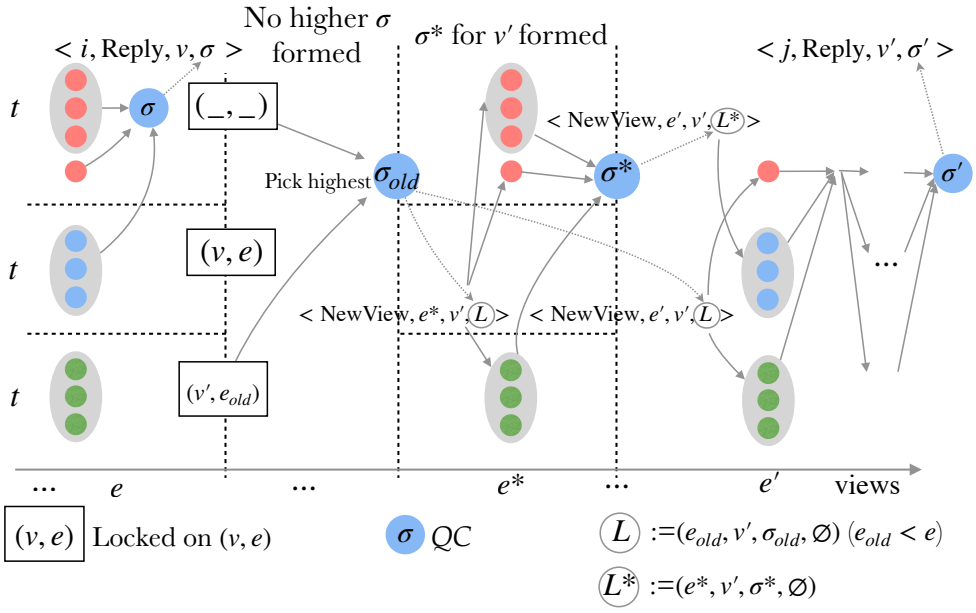


Figure 4.4: World 2 of Theorem 4.5. Replicas are represented as colored nodes. Replica partitions are P (Byzantine), $\{x\}$ (Byzantine), R, and Q from top to bottom.

respectively. Let $commitQC_1$ be on value v in view e, and signed by P, R, x. And let $commitQC_2$ (and a $precommitQC$) be on value v' in view e', and signed by Q, R, x. Suppose the leader of view e, e^*, e' $(e < e^* < e')$ is replica x.

World 1 is presented in Figure 4.3. Let R and x be Byzantine replicas in this world. In view e^*, the leader proposes value v' and Q sends PREPARE on it, but a $prepareQC$ is not formed. In view e', the Byzantine parties, together with Q, sign $prepareQC$ on v'. e' is the first view where a $prepareQC$ for v' is formed.

During the forensic protocol, all honest replicas in P and Q send their transcripts. Byzantine R and x do not provide any information. Since the protocol has forensic support, the forensic protocol can output $d > 1$ replicas in R and x as culprits.

World 2 is presented in Figure 4.4. Let P and x be Byzantine replicas in this world. Here, in view $e^* > e$, P and x, together with Q sign $prepareQC$ on v'. In this world, e^* is the first view where a $prepareQC$ for v' is formed. View $e' > e^*$ is similar to that of World 1 except that honest R receives a NEWVIEW message with $prepareQC^*$ (rather than $prepareQC_{old}$).

During the forensic protocol, Q sends their transcripts, which are identical to those in World 1. Byzantine P can provide the same transcripts as those in World 1. Observe that the transcripts from P and Q presented to the forensic protocol are identical to those in World 1. Thus, the forensic protocol can also outputs $d > 1$ replicas in R and x as culpable.

In World 2, this is incorrect since replicas in R are honest.

Based on $2t$ transcripts, World 1 and World 2 are indistinguishable. To obtain an irrefutable proof of $d > 1$ culprits, the client needs to collect more than $2t$ transcripts, more than the number of honest parties available. QED.

Remark. The above proof can be easily modified to work with parameters $d > 0$ when $m = t + 2$.

4.6 FORENSIC SUPPORT FOR VABA

Validated Asynchronous Byzantine Agreement (VABA) is a state-of-the-art protocol [3] in the asynchronous setting with asymptotically optimal $O(n^2)$ communication complexity and expected $O(1)$ latency for $n \geq 3t + 1$.

4.6.1 Overview

At a high-level, the VABA protocol adapts HotStuff to the asynchronous setting. There are three phases in the protocol:

- **Proposal promotion.** In this stage, each of the n replicas runs n parallel HotStuff-like instances, where replica i acts as the leader within instance i.

- **Leader election.** After finishing the previous stage, replicas run a leader election protocol using a *threshold-coin* primitive [104] to randomly elect the leader of this view, denoted as $Leader[e]$ where e is a view number. At the end of the view, replicas adopt the "progress" from $Leader[e]$'s proposal promotion instance, and discard values from other instances.

- **View change.** Replicas broadcast quorum certificates from $Leader[e]$'s proposal promotion instance and update local variables and/or output value accordingly.

Within a proposal promotion stage, the guarantees provided are the same as that of HotStuff, and hence we do not repeat it here. The leader election phase elects a unique leader at random – this stage guarantees (i) with $\geq 2/3$ probability, an honest leader is elected, and (ii) an adaptive adversary cannot stall progress (since a leader is elected in hindsight). Finally, in the view-change phase, every replica broadcasts the elected leader's quorum certificates to all replicas.

Algorithm 4.6 Forensic protocol for VABA

as a replica running VABA
  for $e \geq 1$ **initialize**:
    for $j \in [n]$ **do**
      $ledger[e][j] \leftarrow \{\}$
      $coin[e] \leftarrow \{\}$
  upon receiving $\langle i, \text{NewView}, e, v, L\rangle$ in view e in Proposal-Promotion instance i **do**
    $(e', v', \sigma, e_{qc}) \leftarrow L$ ▷ Note that L has selectors e, v, σ, e_{qc}
    $ledger[e'][i] \leftarrow ledger[e'][i] \cup \{(v', \sigma, e_{qc})\}$
  upon receiving $\langle i, \text{Prepare}, e, v, \sigma, e_{qc}\rangle$ in view e in Proposal-Promotion instance i **do**
    $ledger[e][i] \leftarrow ledger[e][i] \cup \{(v, \sigma, e_{qc})\}$
  if $Leader[e]$ is elected **then**
    discard $ledger[e][j]$ for $j \neq Leader[e]$
    $coin[e] \leftarrow$ inputs to *threshold-coin* for electing $Leader[e]$
  upon receiving $\langle \text{ViewChange}, e, prepareQC, precommitQC, commitQC\rangle$ in view e **do**
    $(e', v', \sigma, e_{qc}) \leftarrow prepareQC$ ▷ Note that $prepareQC$ has selectors e, v, σ, e_{qc}
    $ledger[e'][Leader[e']] \leftarrow ledger[e'][Leader[e']] \cup \{(v', \sigma, e_{qc})\}$
  upon receiving $\langle \text{Request-Proof-of-Leader}, e, e'\rangle$ from a client **do**
    for all $e \leq e^* \leq e'$ **do**
      send $\langle \text{Proof-of-Leader}, e^*, Leader[e^*], coin[e^*]\rangle$ to client
  upon receiving $\langle \text{Request-Proof}, e, v, \sigma, e'\rangle$ with a collection of $LeaderMsg$ from a client **do**
    for all $e < e^{\#} \leq e'$ **do**
      if $Leader[e^{\#}]$ is not elected yet **then**
        $\langle \text{Proof-of-Leader}, e^{\#}, leader, coin\rangle \leftarrow LeaderMsg$ of view $e^{\#}$
        check *leader* is the leader generated by *coin* in view $e^{\#}$ (otherwise don't reply to the client)
        $Leader[e^{\#}] \leftarrow leader$
      for $qc \in ledger[e^{\#}][Leader[e^{\#}]]$ **do**
        if $(qc.v \neq v) \wedge (qc.e_{qc} \leq e)$ **then**
          send $\langle \text{Proof-across-View}, e^{\#}, Leader[e^{\#}], qc\rangle$ to the client

Algorithm 4.6 (cont.)

as a client
    upon receiving two conflicting REPLY messages **do**
        $e \leftarrow$ the view number of REPLY from lower view
        $e' \leftarrow$ the view number of REPLY from higher view
        for all $e \leq e^* \leq e'$ **initialize**:
            $Leader[e^*] \leftarrow \{\}$
            $LeaderMsg[e^*] \leftarrow \{\}$
        send $\langle$REQUEST-PROOF-OF-LEADER$, e, e'\rangle$ to the replica of REPLY message from higher view
        for all $e \leq e^* \leq e'$ **do**
            wait for $\langle$PROOF-OF-LEADER$, e^*, leader, coin\rangle$ s.t. *leader* is the leader generated by *coin* in view e^* (otherwise the REPLY message is not considered valid)
            $Leader[e^*] \leftarrow leader$
            $LeaderMsg[e^*] \leftarrow \langle$PROOF-OF-LEADER$, e^*, leader, coin\rangle$
        if the two REPLY messages are from different views **then**
            $\langle i,$ REPLY$, e, v, \sigma\rangle \leftarrow$ the message from lower view
            $\langle i',$ REPLY$, e', v', \sigma'\rangle \leftarrow$ the message from higher view
            check $i = Leader[e]$ and $i' = Leader[e']$ (otherwise the REPLY message is not considered valid)
            broadcast $\langle$REQUEST-PROOF$, e, v, \sigma, e'\rangle$ with $LeaderMsg[e^*]$ for all $e < e^* \leq e'$
            wait for $\langle$PROOF-ACROSS-VIEW$, e^\#, leader, qc\rangle$ s.t.
            (1) $e < e^\# \leq e'$, and
            (2) $(qc.v \neq v) \wedge (qc.e_{qc} \leq e)$, and
            (3) $leader = Leader[e^\#]$
            output $qc.\sigma \cap \sigma$
        else
            $\langle i,$ REPLY$, e, v, \sigma\rangle \leftarrow$ first REPLY message
            $\langle i',$ REPLY$, e, v', \sigma'\rangle \leftarrow$ second REPLY message
            check $i = i' = Leader[e]$ (otherwise the REPLY message is not considered valid)
            output $\sigma \cap \sigma'$

4.6.2 Forensic Support for VABA

The key difference between forensic support for HotStuff and VABA is the presence of the leader election stage – every replica/client needs to know *which* replica was elected as the leader in each view. Importantly, the *threshold-coin* primitive ensures that there is a unique leader elected for each view. Thus, the forensic analysis boils down to performing an analysis similar to the HotStuff protocol, except that the leader of a view is described by

the leader election phase.

We present the full forensic protocol in Algorithm 4.6 for completeness. We make the following changes to VABA:

- **Storing information for forensics.** Each replica maintains a list of *ledgers* for all instances, containing all received *prepareQC* from Prepare messages, NewView messages, and ViewChange messages (lines 8,10,16). When the leader of a view is elected, a replica keeps the *ledger* from the leader's instance and discards others (line 12). A replica also stores the random coins from the leader election phase for client verification (line 13).
- **Bringing proposal promotion closer to HotStuff-view.** There are minor differences in the proposal promotion phase of VABA [3] to the description in HotStuff (§4.5). We make this phase similar to that in the description of our HotStuff protocol with forensic support. In particular: (i) the $LOCK$ variable stores both the view number and the value (denoted by $LOCK.e$ and $LOCK.v$), (ii) the voting rule in a proposal promotion phase is: vote if KEY has view and value equal to $LOCK$, except when KEY's view is strictly higher than $LOCK.e$, (iii) assume a replica's own ViewChange message arrives first so that others' ViewChange messages do not overwrite local variables KEY and $LOCK$, and (iv) add e_{qc} into Prepare.

A client first verifies leader election (lines 36-40). Then, it follows steps similar to the HotStuff forensic protocol (lines 41-52) except that there are added checks pertaining to leader elections (lines 44,46,51).

We prove the forensic support in Theorem 4.6.

Theorem 4.6. For $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting values, VABA protocol provides $(2t,\ 1,\ t + 1)$-forensic support. Further $(2t + 1,\ n - f,\ d)$-forensic support is impossible with $d > 0$.

Proof. We prove the forward part of the theorem below. The proof of converse (impossibility) is the same as that of Theorem 4.1.

The leader of a view is determined by the threshold coin-tossing primitive *threshold-coin* and Byzantine replicas cannot forge the result of a leader election by the robustness property of the threshold coin. Suppose two conflicting outputs happen in view e, e' with $e \leq e'$. The replica that outputs in view e' has access to the proof of leader election of all views $\leq e'$. Therefore, a client can verify the leader election when it receives messages from this replica. Even if other replicas have not received messages corresponding to the elections in views $\leq e'$, the client can send the proof of leader to them. The remaining forensic support proof follows from Theorem 4.3 in a straightforward manner, where any witness will receive the

proof of leader from the client (if leader is not elected) and send the proof of culprits to the client. QED.

Communication complexity. The client needs to first receive all leader election results from view e to view e', and each result is of size $|coin|$ (the size of the coin in the *threshold-coin* primitive). Then, the client shares leader election results with all replicas. This step incurs receiving message complexity $O(l|coin|)$ where $l = e' - e$. Next, the client needs to receive one message from $k = 1$ replica and the message size is $(|v| + |sig|)$. Therefore the complexity for the client receiving messages is $O(|v|+|sig|+l|coin|)$. However, the procedure of sharing leader election is irrelevant to forensic support, and we could assign it to replicas. (This procedure is included in the forensic protocol because we do not want to change the consensus protocol itself.) In that case, the client needs to receive just one leader election result, so the receiving message complexity is $O(|v| + |sig| + |coin|)$.

4.7 FORENSIC SUPPORT FOR ALGORAND

Algorand [9] is a synchronous consensus protocol which tolerates up to one-third fraction of Byzantine users. At its core, it uses a BFT protocol from [105, 106]. However, Algorand runs the protocol by selecting a small set of replicas, referred to as the committee, thereby achieving consensus with sub-quadratic communication. The protocol is also player replaceable, i.e., different steps of the protocol have different committees, thus tolerating an adaptive adversary. Each replica uses *cryptographic self-selection* to privately decide whether it is selected in a committee, while cryptographic self-selection works by using a verifiable random function (VRF) [107]. The VRF value is known exclusively to the replica, but after it sends a message to other parties, revealing the value and proving its inclusion in the committee, the value becomes open to everyone. In this section, we present an overview of the BFT protocol and then show why it is impossible to achieve forensic support for this protocol.

4.7.1 Overview

We start with an overview of the single-shot version of Algorand [9], denoted as Algorand (in a sans serif font). The protocol assumes synchronous communication, where messages are delivered within a known bounded time. The protocol proceeds in consecutive steps, each of which lasts for a fixed amount of time that guarantees message delivery. Each step has a self-selected committee, and a replica can compute its VRF value which decides whether

it is selected in the committee. The VRF value is known exclusively to the replica itself before it sends the value. All messages sent by a committee member is accompanied by the VRF which allows other replicas to verify its inclusion in the committee. Parameters such as committee size κ are chosen such that the number of honest parties in the committee is greater than a threshold $t_H \geq 2\kappa/3$ with overwhelming probability.

The BFT protocol is divided into two sub-protocols: *Graded Consensus* and BBA^*. In *Graded Consensus* which forms the first three steps of the protocol, each replica r inputs its value v_r. Each replica r outputs a tuple containing a value v_r^{out} and a grade g_r. In an execution where all replicas start with the same input v, $v_r^{\text{out}} = v$ and $g_r = 2$ for all replicas r. The replicas then enter the next sub-protocol, denoted BBA^*. If $g_r = 2$, replica r inputs value $b_r = 0$, otherwise it inputs $b_r = 1$. At the end of BBA^*, the replicas agree on the tuple $(0, v)$ or $(1, v_\perp)$.[3] The BBA^* sub-protocol also uses a random coin. For simplicity, we assume the access to an ideal global random coin. Our forensic analysis in the next section only depends on BBA^* and thus, we only provide an overview for BBA^* here. The protocol proceeds in the following steps,

- Steps 1-3 are *Graded Consensus*. At the end of *Graded Consensus*, each replica inputs a value v_r and a binary value b_r to BBA^*.

- Step 4. Each replica in the committee broadcasts its vote for (b_r, v_r) along with its VRF.

- Step s ($s \geq 5, s \equiv 2 \mod 3$) is the Coin-Fixed-To-0 step of BBA^*. In this step, a replica checks *Ending Condition 0*: if it has received $\geq t_H$ valid votes on (b, v) from the previous step, where $b = 0$, it outputs (b, v) and ends its execution. Otherwise, it updates b_r as follows:

$$b_r \leftarrow \begin{cases} 1, & \text{if} \geq t_H \text{ votes on } b = 1 \\ 0, & \text{otherwise} \end{cases} \tag{4.1}$$

  If the replica is in the committee based on its VRF, it broadcasts its vote for (b_r, v_r) along with the VRF.

- Step s ($s \geq 6, s \equiv 0 \mod 3$). Symmetric to the previous step but for bit 1 instead of 0. Also, the votes need not be for the same v in the ending condition.

- Steps s ($s \geq 7, s \equiv 1 \mod 3$) is the Coin-Genuinely-Flipped step of BBA^*. In this

[3] $v_\perp$ is considered external valid in Algorand.

step, it updates its variable b_r as follows:

$$b_r \leftarrow \begin{cases} 0, \text{if} \geq t_H \text{ votes on } b = 0 \\ 1, \text{if} \geq t_H \text{ votes on } b = 1 \\ \text{random coin of step } s, \quad \text{otherwise} \end{cases} \tag{4.2}$$

If the replica is in the committee based on its VRF, it broadcasts its vote for (b_r, v_r) along with the VRF.

Safety of BBA^* within a step. If all honest replicas reach an agreement before any step, the agreement will hold after the step. If the agreement is on binary value 0 (1 resp.) then the opposite Ending Condition 1 (0 resp.) will not be satisfied during the step. This is because synchronous communication ensures the delivery of at least t_H votes on the agreed value and there are not enough malicious votes on the other value.

Safety of BBA^* across steps. For the step Coin-Fixed-To-0 (1 resp.), if any honest replica ends due to Ending Condition 0 (1 resp.), all honest replicas will agree on binary value 0 and value v (1 and $v_\perp$ resp.) at the end of the step, because there could only be less than t_H votes on binary value 1 (0 resp.). Hence, together with safety within a step, binary value 1 and value $v_\perp$ (0 and v resp.) will never be output.

4.7.2 Impossibility of Forensics

When the Byzantine fraction in the system is greater than one-third, with constant probability, a randomly chosen committee of size $\kappa < n$ will have $t_H < 2\kappa/3$. In such a situation, we can have a safety violation. Observe that since only $\kappa < n$ committee members send messages in a round, the number of culpable replicas may be bounded by $O(\kappa)$. However, we will show an execution where no Byzantine replica can be held culpable.

Intuition. The safety condition for BBA^* relies on the following: if some honest replica commits to a value b, say $b = 0$, in a step and terminates, then all honest replicas will set $b = 0$ as their local value. In all subsequent steps, there will be sufficient ($> 2/3$ fraction) votes for $b = 0$ due to which replicas will never set their local value $b = 1$. Thus, independent of what Byzantine replicas send during the protocol execution, honest replicas will only commit on $b = 0$. On the other hand, if replicas do not receive $> 2/3$ fraction of votes for $b = 0$, they may switch their local value to $b = 1$ in the Coin-Fixed-To-1 or Coin-Genuinely-Flipped step. This can result in a safety violation. When the Byzantine fraction

is greater than one-third, after some replicas have committed 0, the Byzantine replicas can achieve the above condition by selectively not sending votes to other replicas (say set Q), thereby making them switch their local value to $b = 1$. There is no way for an external client to distinguish this world from another world where the set Q is Byzantine and states that it did not receive these votes. We formalize this intuition in the theorem below. Observe that our arguments work for the BBA^* protocol with or without player replaceability.

Theorem 4.7. When the Byzantine fraction exceeds 1/3, if two honest replicas output conflicting values, $(t + 1,\ 2t,\ d)$-forensic support is impossible with $d > 0$ for Algorand.

Proof. We construct two worlds where a different set of replicas is Byzantine in each world. Let replicas be split into three partitions P, Q, and R, and $|P| = (n - 2\epsilon)/3, |Q| = |R| = (n + \epsilon)/3$ and $\epsilon > 0$ is a small constant. Denote the numbers of replicas from P, Q, R in a committee by p, q, r. Let κ denote the expected committee size; $t_H = 2\kappa/3$. With constant probability, we will have $p < \kappa/3$, $q > \kappa/3$ and $r > \kappa/3$ and $p + q < 2\kappa/3$ in steps 4 to 8.

World 1. Replicas in R are Byzantine in this world. We have $p + q < t_H$ and $q + r > t_H$. The Byzantine parties follow the protocol in *Graded Consensus*. Thus, all replicas in step 4 hold the same tuple of $b = 0$ and v $(v \neq v_\perp)$. Then, the following steps are executed.

- Step 4. Honest committee members that belong to P and Q broadcast their votes on $(b = 0, v)$ whereas Byzantine committee members that belong to R send votes to replicas in P and not Q.
- Step 5. Replicas in P satisfy Ending Condition 0, and output $b = 0$ and the value v. Replicas in Q do not receive votes from committee members in R, so they update $b = 0$ and broadcast their votes on $(b = 0, v)$. Byzantine committee members that belong to R pretend not to receive votes from committee members in Q, and also update $b = 0$. And they send votes to replicas in P and not Q.
- Step 6. Replicas in Q update $b = 1$ since they receive $p + q < t_H$ votes. Replicas in R pretend not to receive votes from committee members in Q, and also update $b = 1$. Committee members in Q and R broadcast their votes.
- Steps 7-8. Committee members that belong to Q and R receive $q + r > t_H$ votes, so they update $b = 1$ and broadcast their votes.
- Step 9. Replicas in Q and R satisfy Ending Condition 1, and output $b = 1$ and $v_\perp$, a disagreement with replicas in P.

During the forensic protocol, replicas in P send their transcripts and state that they have output $b = 0$. Q and R send their transcripts claiming in steps 4 and 5 they do not hear from the other partition, and they state that output $b = 1$.

If this protocol has any forensic support, then it should be able to detect some replica in R as Byzantine.

World 2. This world is identical to World 1 except (i) Replicas in Q are Byzantine and replicas in R are honest, and (ii) the Byzantine set Q behaves exactly like set R in World 1, i.e., replicas in Q do not send any votes to R in steps 4 and 5 and ignore their votes. During the forensic protocol, P send their transcripts and state that they have output $b = 0$. Q and R send their transcripts claiming in steps 4 and 5 they do not hear from the other partition, and they state that output $b = 1$.

From an external client's perspective, World 2 is indistinguishable from World 1. In World 2, the client should detect some replica in R as Byzantine as in World 1, but all replicas in R are honest. QED.

4.8 FORENSIC SUPPORT FOR DIEMBFT

In this chapter, we have focused on forensics for single-shot consensus. Chained BFT protocols are natural candidates for consensus on a sequence with applications to blockchains. DiemBFT is a chained version of HotStuff and is the core consensus protocol in Diem, a new cryptocurrency supported by Facebook [108]. In this section, we show that DiemBFT has the strongest forensic support possible (as in HotStuff-view). Further, we implement the corresponding forensic analysis protocol as a module on top of an open source Diem client. We highlight the system innovations of our implementation and the associated forensic dashboard.

Diem blockchain. Diem Blockchain uses DiemBFT [10], a chained variant of HotStuff protocol for consensus. In DiemBFT, the replicas are called validators, who receive transactions from clients and propose blocks of transactions in a sequence of rounds.

DiemBFT forensics. The culpability analysis for DiemBFT is similar to Theorem 4.4. However, for the witnesses, the blockchain property of DiemBFT makes sure that any replica (validator) has access to the full blockchain and thus provides $(n-2, 1, t+1)$-forensic support. The formal result is below.

Theorem 4.8. For $n = 3t + 1$, when $f > t$, if two honest replicas output conflicting blocks, DiemBFT provides $(n - 2,\ 1,\ t + 1)$-forensic support.

Proof. Suppose two conflicting blocks $b,\ b'$ are output in views $e,\ e'$ respectively.

Case $e = e'$. Culpability. The *commitQC* of b (the QC in $e+3$) and *commitQC* of b' intersect in $t+1$ replicas. These $t+1$ replicas should be Byzantine since the protocol requires a replica to vote for at most one value in a view.
Witnesses. Client can get the proof based on the two blocks in $e+3$, so additional witnesses are not necessary in this case.

Case $e \neq e'$. Culpability. If $e \neq e'$, then WLOG, suppose $e < e'$. Since b is output in view e, it must be the case that $2t+1$ replicas are locked on (b, e) at the end of view e. Now consider the first view $e < e^* \leq e'$ in which a higher lock (b'', e^*) is formed where b'', b are not on the same chain (possibly b'' is on the chain of b'). Such a view must exist since b' is output in view $e' > e$ and a lock will be formed in at least view e'. For a lock to be formed, a higher *prepareQC* must be formed too.

Consider the first view $e < e^{\#} \leq e'$ in which a *prepareQC* in chain of b'' is formed. The leader in $e^{\#}$ broadcasts the block containing a *highQC* on (b'', e''). Since this is the first time a higher *prepareQC* is formed and there is no *prepareQC* for chain of b'' formed between view e and $e^{\#}$, we have $e'' \leq e$. The formation of the higher *prepareQC* indicates that $2t+1$ replicas received the block extending b'' with *highQC* on (b'', e'') and consider it a valid proposal, i.e., the view number e'' is larger than their locks because the block is on another chain.

Recall that the output block b indicates $2t+1$ replicas are locked on (b, e) at the end of view e. In this case, the $2t+1$ votes in *prepareQC* in view $e^{\#}$ intersect with the $2t+1$ votes in *commitQC* in view e at $t+1$ Byzantine replicas. These replicas should be Byzantine because they were locked on the block b in view e and vote for a conflicting block in a higher view $e^{\#}$ whose *highQC* is from a view $e'' \leq e$. Thus, they have violated the voting rule.
Witnesses. Client can get the proof by storing a *prepareQC* formed in $e^{\#}$ between e and e' in a different chain from b. The *prepareQC* is for the previous block in $e^{\#}$ whose *highQC* is formed in a view $e'' < e$. For the replicas who have access to the *prepareQC*, they must have access to all blocks in the same blockchain. Thus, only one witness is needed ($k = 1$) to provide the *prepareQC* and its previous block containing the *highQC* on (b'', e''). The *prepareQC*, the *highQC*, and the first *commitQC* act as the irrefutable proof. QED.

The aforementioned three variants of HotStuff in §4.5 are described under the BA (single-shot) setting to reach consensus on a single value which can be directly included in the vote message (cf. Table 4.4, v is contained in the PREPARE message). In this setting, once a replica receives the *commitQC* for the value, it will output the value and send a reply to the client, even if the *commitQC* is the only message it receives in the current view so far. So

when two honest replicas output conflicting values, it is possible that the client receives only the commit messages and extra communication is needed. And when $m > 2t$, Byzantine replicas are able to form QCs by themselves so that no other honest replicas can get access to the first *prepareQC*. Thus the bound on m for HotStuff-view and HotStuff-hash is $2t$.

However, the setting is slightly different in practice when the value v is no longer a single value, but actually a block with more fields and a list of transactions/commands, as in DiemBFT. In single-shot consensus, a block includes the transactions (value) and the *highQC*. In this case, the block is too heavy to be included in a vote message, so the replicas add the hash of the block to the vote message (see Table 4.4, $\text{Hash}(b = (v, highQC))$ is contained in the PREPARE message). And since only the NEWVIEW message has the block's preimage, replicas cannot vote/output until receiving the original blocks. Thus when two honest replicas output conflicting values, the client can obtain the full blockchain from one of them ($k = 1$) and all *prepareQC* are part of the blocks. In this case, even if $m > 2t$ the client can still enjoy non-trivial forensic support.

Table 4.4: Comparison of HotStuff-hash and DiemBFT

	HotStuff-hash	DiemBFT
PREPARE	$\langle \text{PREPARE}, e, v, \text{Hash}(highQC) \rangle$	$\langle \text{PREPARE}, e, \text{Hash}(b = (v, highQC)) \rangle$
k	$t + 1$	1
m	$2t$	$n - 2$
extra condition	-	must receive the preimage of hash

Forensic module. Our prototype consists of two components, a database FORENSIC STORAGE used to store quorum certificates received by validators, which can be accessed by clients through JSON-RPC requests or consensus API; and an independent DETECTOR run by clients to analyze the forensic information.

- **Forensic Storage** maintains a map from the view number to quorum certificates and stores it persistently. It is responsible for storing forensic information and allows access by other components, including clients (via JSON-RPC requests or consensus API).
- **Detector** is run by clients manually to send requests periodically to connected validators. It collates information received from validators, using it as the input to the forensic analysis protocol.

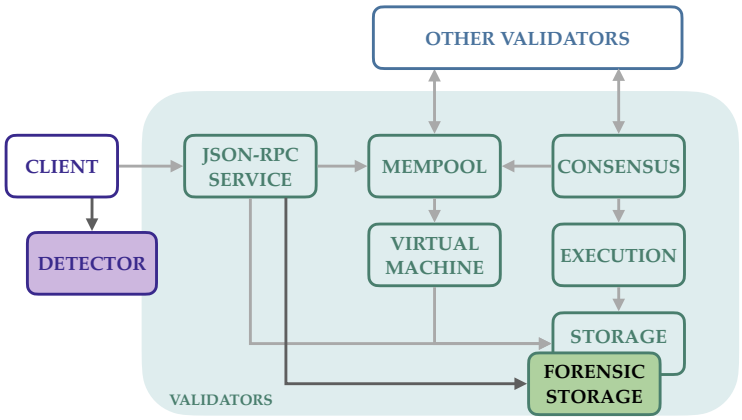


Figure 4.5: Forensic module integrated with Diem.

Testing using Twins [109]. To test the correctness of forensic protocols, we build a testbed to simulate Byzantine attacks and construct different types of safety violations. Ideally, for modularity purposes, our testbed should not require us to modify the underlying consensus protocol to obtain Byzantine behavior. We leverage Twins [109], an approach to emulate Byzantine behaviors by running two instances of a node (i.e. replica) with the same identity. Consider a simple example setting with four nodes (denoted by $node0 \sim 3$), where $node0$ and $node1$ are Byzantine so they have twins called $twin0$ and $twin1$. The network is split into two partitions, the first partition P_1 includes nodes $\{node0, node1, node2\}$ and the second partition P_2 includes nodes $\{twin0, twin1, node3\}$. Nodes in one partition can only receive the messages sent from the same partition. The double voting attack can be simulated when Byzantine leader proposes different valid blocks in the same view, and within each partition, all nodes will vote for the proposed block. The network partition is used to drop all messages sent from a set of nodes. However, it can only help construct the safety violation within the view. To construct more complicated attacks, we further improve the framework and introduce another operation called "detailed drop", which drops selected messages with specific types.

Visualization. The DETECTOR accepts the registration of different views to get notified once the data is updated. We built a dashboard to display the information received by the detector and the analysis results output by the forensic protocol. Figure 4.6 shows a snapshot of the dashboard which displays information about the network topology, hashes of latest blocks received at different validators, conflicting blocks, detected culprit keys and raw logs. Interactions with end-users, including Diem core-devs, have guided our design of the dashboard.

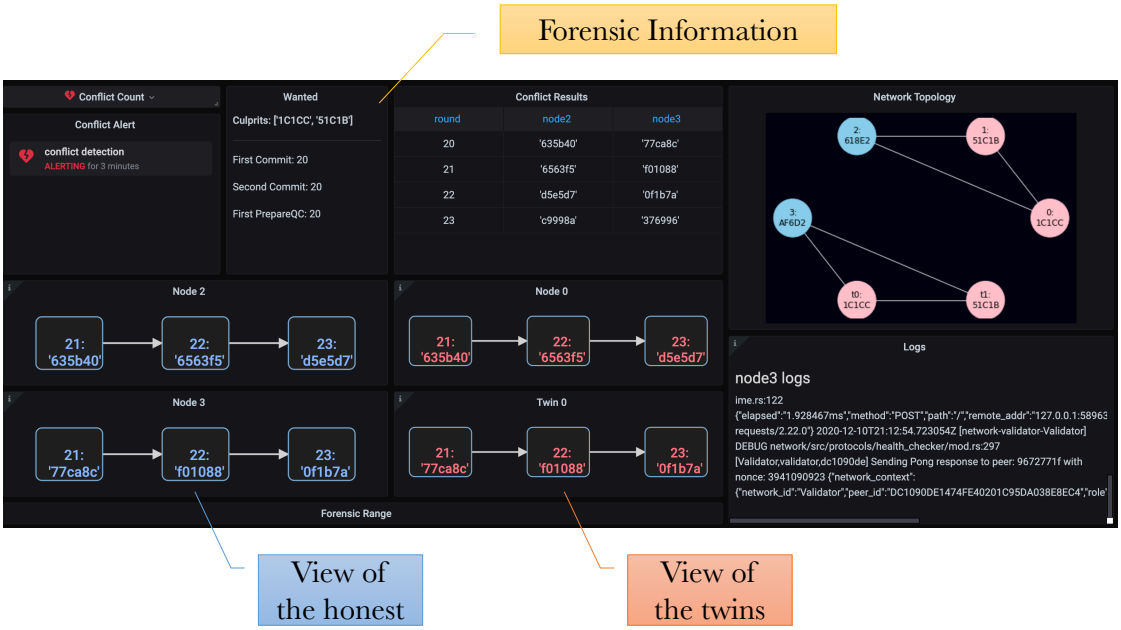


Figure 4.6: Forensic module dashboard.

4.9 IMPOSSIBILITY OF FORENSIC SUPPORT FOR $N = 2T + 1$

A validated Byzantine agreement protocol allows replicas to obtain agreement, validity, and termination so far as the actual number of faults $f \leq t$ where t is a Byzantine threshold set by the consensus protocol. A protocol that also provides forensic support with parameters m and d allows the detection of d Byzantine replicas when $\leq m$ out of n replicas are Byzantine faulty. In particular, in §4.4 and §4.5, we observed that when $t = \lfloor n/3 \rfloor$, $m = 2t$, and $k = 1$, we can obtain $(2t, 1, d)$-forensic support for $d = t + 1$. This section presents the limits on the number of Byzantine replicas detected (d), given the total number of Byzantine faulty replicas available in the system (m). In particular, we show that if the total number of Byzantine faults are too high, in case of a disagreement, the number of corrupt (Byzantine) replicas that can be deemed undeniably culpable will be too few.

Intuition. To gain intuition, let us consider a specific setting with $n = 2t+1$, $m = n-t = t+1$, and $d > 1$. Thus, such a protocol provides us with agreement, validity, and termination if the Byzantine replicas are in a minority. If they are in the majority, the protocol transcript provides undeniable guilt of more than one Byzantine fault. We show that such a protocol does not exist. Why? Suppose we split the replicas into three groups P, Q, and R of sizes t, t, and 1 respectively. First, observe that any protocol cannot expect Byzantine replicas to participate in satisfying agreement, validity, and termination. Hence, if the replicas in Q are Byzantine, replicas in $P \cup R$ may not receive any messages from Q. However, if, in

addition, the replica R is also corrupt, then $R \cup Q$ can separately simulate another world where P is Byzantine and not sending messages, and $Q \cup R$ outputs a different value. Even if an external client obtains a transcript of the entire protocol execution (i.e., transcripts of $k = n - f$ honest replicas and f Byzantine replicas), the only replica that is undeniably culpable is R since it participated in both worlds. For all other replicas, neither P nor Q has sufficient information to prove the other set's culpability. Thus, an external client will not be able to detect more than one Byzantine fault correctly. Our lower bound generalizes this intuition to hold for $n > 2t$, $m = n - t$, $k = n - f$, and $d > n - 2t$.

Theorem 4.9. For any validated Byzantine agreement protocol with $t < n/2$, when $f > t$, if two honest replicas output conflicting values, $(n-t,\ n-f,\ d)$-forensic support is impossible with $d > n - 2t$.

Proof. Suppose there exists a protocol that achieves agreement, validity, termination, and forensic support with parameters $n,\ t < n/2,\ m = n-t,\ k = n-f$ and $d > n-2t$. Through a sequence of worlds, and through an indistinguishability argument we will show the existence of a world where a client incorrectly holds at least one honest replica as culpable. Consider the replicas to be split into three groups $P,\ Q$, and R with t, t, and $n-2t$ replicas respectively. We consider the following sequence of worlds:

World 1. [t Byzantine faults, satisfying agreement, validity, and termination]
Setup. Replicas in P and R are honest while replicas in Q have crashed. P and R start with a single externally valid input v_1. All messages between honest replicas arrive instantaneously.
Output. Since there are $|Q| = t$ faults, due to agreement, validity and termination properties, replicas in P and R output v_1. Suppose replicas in P and R together produce a transcript T_1 of all the messages they have received.

World 2. [t Byzantine faults, satisfying agreement, validity, and termination]
Setup. Replicas in Q and R are honest while replicas in P have crashed. Q and R start with an externally valid input v_2. All messages between honest replicas arrive instantaneously.
Output. Since t replicas are Byzantine faulty, due to agreement, validity and termination properties, replicas in Q and R output v_2. Suppose replicas in Q and R together produce a transcript T_2 of all the messages they have received.

World 3. [$n - t$ Byzantine faults satisfying validity, termination, forensic support]
Setup. Replicas in P are honest while replicas in Q and R are Byzantine. Replicas in P start with input v_1. Replicas in Q and R have access to both inputs v_1 and v_2. Q behaves as if it

starts with input v_2 whereas R will use both inputs v_1 and v_2. Replicas in Q and R behave with P exactly like in World 1. In particular, replicas in Q do not send any message to any replica in P. Replicas in R perform a split-brain attack where one brain interacts with P as if the input is v_1 and it is not receiving any message from Q. Also, separately, replicas in Q and the other brain of R start with input v_2 and communicate with each other exactly like in World 2. They ignore messages arriving from P.
Output. For replicas in P, this world is indistinguishable from that of World 1. Hence, they output v_1. Replicas in P and the first brain of R output transcript T_1 corresponding to the output. Replicas in Q and the other brain of R behave exactly like in World 2. Hence, they can output transcript T_2. Since the protocol provides $(n-t, n-f, d)$-forensic support for $d > n-2t$, the transcript of messages should hold $d > n-2t$ Byzantine replicas undeniably corrupt. Suppose the client can find the culpability of $> n-2t$ replicas from $Q \cup R$, i.e., ≥ 1 replica from Q.

World 4. [$n-t$ Byzantine faults satisfying validity, termination, forensic support]
Setup. Replicas in Q are honest while replicas in P and R are Byzantine. Replicas in Q start with input v_2. Replicas in P and R have access to both inputs v_1 and v_2. P behaves as if it starts with input v_1 whereas replicas in R use both v_1 and v_2. Replicas in P and R behave with Q exactly like in World 2. In particular, replicas in P do not send any message to any replica in Q. Replicas in R perform a split-brain attack where one brain interacts with Q as if the input is v_2 and it is not receiving any message from P. Also, separately, replicas in P and the other brain of R start with input v_1 and communicate with each other exactly like in World 1. They ignore messages arriving from Q.
Output. For replicas in Q, this world is indistinguishable from that of World 2. Hence, they output v_2. Replicas in Q and the first brain of R output transcript T_2 corresponding to the output. Replicas in P and the other brain of R behave exactly like in World 1. Hence, they can output transcript T_1.

Observe that the transcript and outputs produced by replicas in P, Q, and R are exactly the same as in World 3. Hence, the client will hold $> n-2t$ replicas from $Q \cup R$, i.e., ≥ 1 replica from Q as culpable. However, all replicas in Q are honest in this world. This is a contradiction. QED.

4.10 CONCLUSION

We have embarked on a systematic study of the forensic properties of BFT protocols, focusing on 4 canonical examples: PBFT (classical), Hotstuff and VABA (state-of-the-art

protocols on partially synchronous and asynchronous network settings) and Algorand (popular protocol that is adaptable to proof-of-stake blockchains). Our results show that minor variations in the BFT protocols can have outsized impact on their forensic support.

We exactly characterize the forensic support of each protocol, parameterized by the triplet (m, k, d). The forensic support characterizations are remarkably similar across the protocols: if any non-trivial support is possible (i.e., at least one culpable replica can be implicated; $d > 0$), then the largest possible forensic support, $(2t, 1, t + 1)$, is also possible. The one exception to this result is the HotStuff-hash variant. Although the proof of forensic support is conducted for each protocol and its variant individually, we observe common trends:

- For each of the protocols with strong forensic support, as a part of the protocol execution, there exist witnesses who hold signed messages from Byzantine parties indicating that they have not followed some rule in the protocol.
- On the other hand, for protocols with no forensic support, the Byzantine parties are able to break safety without leaving any evidence, although the mechanism to achieve this is different for each of PBFT-MAC, Algorand, and HotStuff-null. With PBFT-MAC, Byzantine parties are able to construct arbitrary transcripts due to the absence of signatures. Hence, message transcripts cannot be used as evidence. With Algorand, they can utilize a rule which relies on the absence of messages (under synchrony) to set an incorrect protocol state without leaving a trail. With HotStuff-null, due to the lack of links between messages across views, Byzantine parties can present fake message transcripts and thus, pretend to be honest.

Conceptually, the burning question is whether these common ingredients can be stitched together to form an overarching theory of forensic support for abstract families of secure BFT protocols: First, from an impossibility standpoint, is there a relationship between the need to use synchrony or the absence of PKI in a protocol and absence of forensic support? Second, for the positive results, can one argue strong forensic support for an "information-complete" variant of any BFT protocol? This is an active area of research.

From a practical standpoint, forensic analysis for existing blockchain protocols is of great interest to the industry. Our forensic protocol for DiemBFT and its reference implementation has made strong inroads towards practical deployment. However, one shortcoming of the approach in this chapter is that forensic analysis is conducted only upon *fatal* safety breaches. It is of great interest to conduct forensics with other forms of attacks: liveness attacks, censorship, a small number of misbehaving replicas that impact performance. This is an active area of research.

CHAPTER 5: PLAYER REPLACEABLE BLOCKCHAINS WITH FORENSIC SUPPORT

Player replaceability is a property of a blockchain protocol that ensures every step of the protocol is executed by an unpredictably random set of players. It guarantees security against a fully adaptive adversary. *Forensic Support* is a property of a blockchain protocol that provides the ability, with cryptographic integrity, to identify malicious parties when there is a safety violation. It provides the ability to enforce punishments for adversarial behavior and is a crucial component of incentive mechanism designs for blockchains. Player replaceability and strong forensic support are both desirable properties, yet, none of the existing BFT blockchain protocols have *both* properties. In this chapter, we construct a new BFT protocol which is player replaceable and has maximum forensic support. The key invention is the notion of a "transition certificate", without which we show that natural adaptation of extant BFT protocol does not lead to the desired goal of simultaneous player replaceability and forensic support.

In addition, we adapt the notion of a transition certificate for another family of protocols, reconfigurable protocols, which both keep a fixed set of participants within an epoch (a period of time) and reconfigure a new set of players when it comes to the next epoch. As a result, forensic support during the epoch switch is enhanced.

We give an introduction to this work in §5.1. We describe the security model and definitions in §5.2. §5.3 contains the construction of a new BFT protocol endowed with both player replaceability and strong forensic support. §5.4 extends the result to reconfigurable protocols. Related works that are not covered in the previous sections are discussed in § 5.5. §5.6 concludes the chapter with a discussion of the relationship between player replaceability and forensic support.

This chapter is a joint work with Peiyao Sheng, Kartik Nayak, Sreeram Kannan, and Pramod Viswanath.

5.1 INTRODUCTION

Byzantine fault-tolerant state machine replication (BFT SMR) protocols allow a group of parties to agree on a common sequence of values submitted by external clients. The core security guarantee provided by BFT SMR is that as long as a certain fraction of parties are honest, i.e., follow the protocol, then these parties achieve consensus with respect to a time-evolving ledger regardless of the actions of the remaining malicious (Byzantine) parties that deviate from the protocol. Of particular interest are secure and efficient BFT SMR

protocols: security is measured via tolerating the maximum number of Byzantine parties under various network and cryptographic assumptions [5, 110, 111, 112, 113, 114, 115], and efficiency is measured in terms of commit latency and communication complexity [2, 14, 112, 116, 117, 118].

Security guarantee of BFT protocols is one-sided, addressing the scenario when the number of Byzantine parties is less than a certain threshold. *Forensic support* addresses the other side: what happens when the number of Byzantine parties exceeds the allowable threshold? Several recent works focus on designing secure BFT protocols that also have an additional goal of accountability, i.e., the ability to detect faulty behavior through an irrefutable proof upon security violation [46, 96, 98, 99, 119]. A recent work [89] has formally defined forensic support of BFT protocols, providing a unified framework to compare and contrast different designs. It also provides a detailed analysis of canonical BFT protocols (e.g., PBFT [7, 8], HotStuff [2], VABA [3], and Algorand [9, 21, 106]) with respect to their support for forensics on detecting Byzantine behavior. A key takeaway from this work is that, while forensic support depends heavily on the implementation details of the protocol, deterministically secure protocols with $\mathsf{poly}(n)$ communication complexity (here, n is the number of parties in the protocol) have protocol variants with maximum forensic support (i.e., the maximum number of Byzantine parties can be identified irrefutably using simply the transcript available at one of the honest parties).

An entirely different aspect of BFT protocols has emerged with the advent of blockchains and the desire to support the participation of a vast number of players: communication-efficiency (i.e., have sub-quadratic communication complexity) combined with security against a fully adaptive adversary; for example, Algorand [106]. Such protocols are commonly referred to as "player replaceable" since they rely on verifiably selecting small subgroups of truly random parties in each round, thus achieving adaptive security and communication efficiency. Of specific interest are secure blockchain protocols that offer both desired properties: forensic support *and* player replaceability. We begin by observing that no extant blockchain protocols offer both strong forensic support and player replaceability. For instance, HotStuff excels in efficiency and forensic support but is not player replaceable; Algorand is player replaceable but has non-existent forensic support [89]. Indeed, no extant blockchain protocol appears to have both player replaceability and strong forensic support.

Another class of protocols that substitute protocol participants is "reconfigurable" protocols, in which they select a subgroup of parties and keep them fixed for the duration of an epoch. Reconfigurable protocols have a relaxed selection of players in that they have no requirement on random and each-step selection. They are not secure against adaptive adversaries as player replaceable ones do, yet they provide flexibility in substituting players

in that designers can choose specific reconfiguration rules. The example of these protocols includes DiemBFT [10] which designs configuration-change commands and plans to use them for epoch switch.

Exploring whether there is a contradiction between player replaceability and forensic support properties of BFT protocols is the main goal of this chapter. Our main result is clear: we construct a new BFT protocol that is player replaceable *and* has strong forensic support (i.e., detecting the maximum number of Byzantine nodes with the minimum number of honest transcripts). Meanwhile, we show that natural adaptations of extant BFT protocols do not lead to the desired goal. Next, we extend this protocol to build a framework for reconfigurable protocols to obtain strong forensic support. A summary of our results is presented in Table 5.1.

Table 5.1: Comparison of forensic properties among different protocols

	Protocol	**Byzantine threshold (t)**	**Player replaceability**	**Forensic support (d)**	
BFT Protocols	Algorand [9]	$n/3$	Yes	None	0
	HotStuff [2]		No	Strong	$\lceil n/3 \rceil$
	Algorithm 5.3		Yes	None	0
Our Result	**Algorithm 5.1**	$n/3$	**Yes**	**Strong**	$\lceil \lambda/3 \rceil$[③]

[①] λ is the size of committee

Main result: a player replaceable BFT protocol with strong forensic support. We first present a novel player replaceable BFT protocol with strong forensic support in the partially synchronous setting, where "strong" implies that most number of Byzantine nodes can be detected with the least number of honest transcripts. In particular, we show that when the total fraction of Byzantine parties are fewer than $(1-\epsilon)2/3$ (ϵ is a positive constant) and committee sizes are λ, our forensic protocol can detect at least $\lceil \lambda/3 \rceil$ Byzantine parties. Due to idiosyncratic constraints imposed by player replaceability, traditional analyses of forensic support [89] do not immediately apply. For instance, a core component of the forensic support analysis of existing BFT protocols relies on identifying parties that perform two or more actions that are incompatible with each other with respect to the protocol specification [89]. However, with player replaceability, when n is large, it is extremely unlikely that the same player will be selected twice; thus access to incompatible actions performed by the same player, especially across different rounds (or views) of the protocol, is unavailable. One of our key innovations is the notion of "transition certificates", maintained and shared

by each party in each round – this ensures that if Byzantine parties vote incorrectly in a round resulting in a safety violation, there is sufficient information to detect misbehavior.

While there exist player replaceable BFT protocols with forensic support, it turns out that minor implementation details matter to achieving both properties simultaneously. To highlight this subtlety, we construct a protocol that is only slightly different from the one in our main result – the only difference being the absence of sharing and maintenance of transition certificates. Through an indistinguishability argument, we show that this player replaceable protocol has no forensic support even though it is secure when the Byzantine threshold is within bounds.

The result extension: adapt reconfigurable protocols for strong forensic support. We take the critical innovation of transition certificates and apply it to a framework of reconfigurable protocols. The difficulty for forensic analysis of reconfigurable protocols lies within the epoch switch period, i.e., the steps during the substitution of parties. This adaptation enables strong forensic support during epoch switch periods. If the choice of the original protocol has strong forensic support within an epoch when the parties are fixed, this adaptation has strong forensic support for the whole protocol execution.

5.2 MODEL AND DEFINITIONS

We consider a network with n nodes interacting via all-to-all communication. Prior to the protocol execution, each node generates its public / private key pair honestly and sends its public key to all other nodes. The adversary can adaptively corrupt nodes at any time during the protocol execution after the trusted setup. Nodes that are never corrupted are referred to as honest. The total number of nodes corrupted by the adversary in an execution is denoted as f. The maximum number of corrupted nodes the protocols can tolerate is denoted as t.

Network setting. We consider a partially synchronous network setting. In a partially synchronous protocol, there exists an unknown global stabilization time (GST), after which all transmissions between two honest nodes arrive within a bounded network delay Δ [5].

Blockchains and state machine replication (SMR). The goal of blockchains (state machine replication) is to build a public ledger that provides clients a totally ordered sequence of transactions. The key security properties a blockchain protocol should provide are those

of safety and liveness.[1]

- Safety: no two honest nodes finalize two different blocks at the same position in the ledger.
- Liveness: every valid transaction is eventually finalized by every honest node.

We use blockchain and SMR interchangeably and refer to nodes that run blockchain protocols as "replicas", "players", or "participants".

Player replaceability is a property of blockchain protocols: each step of the protocol execution is conducted by an independently and randomly selected subset of players. Equivalently, player replaceable protocols are those that are secure even in the presence of a fully adaptive adversary. We are especially interested in efficient player replaceable protocols, those with communication complexity subquadratic in number of players.

Reconfigurability is a property of blockchain protocols: an epoch is a period of times or protocol steps, and each epoch of the protocol execution is conducted by a newly selected subset of players, whereas within an epoch, the players are fixed and keep running the protocol. Protocol designers specify the selection of players. They have the flexibility to include diversified factors, such as stakes, performance, or external appointment, which is out of the scope of this work.

Forensic support for blockchains. The notion of forensic support for Byzantine Agreement (BA) was introduced by [89]. Forensic support refers to the ability to identify misbehaving replicas whenever there is a safety violation (two honest replicas finalize different blocks at the same position). The number of replicas that can be held culpable when $t < f \leq m$ is captured by the parameter d (where m is the maximum number of Byzantine replicas under which the forensic support can be provided). Thus, the protocol has forensic support only when the number of faults is fewer than m. In BA, transcripts of honest parties are needed to obtain irrefutable proof of culprits *after* clients detect a safety violation. The number of transcripts to decide culpability of replicas is denoted by k. In the blockchain setting, we adapt the definition of k to denote the number of transcripts required to detect safety violations and construct the culpability proof.

Definition 5.1. $(\mathbf{m}, \mathbf{k}, \mathbf{d})$**-Forensic Support**. If $t < f \leq m$ and there is a safety violation, then using the transcripts of all messages received from k honest replicas during the protocol, a client can provide an irrefutable proof of culpability of at least d Byzantine replicas.

[1]Compared to the definition in Chapter 4, "validity" is simplified in liveness property as "...valid transaction..." since violation of validity is easy to detect.

Cryptographic primitives. All protocols we discuss here use collision resistant cryptographic hash functions and digital signatures. $\langle m \rangle$ denotes the signed message m. The intersection of two aggregated signatures refers to the set of replicas who sign both messages. We use verifiable random functions (VRFs) [107] to choose a random subset of replicas to be the leader or committee in a round. In our model, VRF has two functions: $\texttt{VRF}_{sk}(x)$ and $\texttt{VerifyVRF}_{pk}(msg, x)$. $\texttt{VRF}_{sk}(x)$ returns two values: a hash $hash$ and a proof π. The hash is a $HASHLEN$-bit value, normalized by $2^{HASHLEN}$, i.e., $hash \in [0, 1]$. It is uniquely determined by sk and x, and indistinguishable from a random value to anyone that does not know sk. The proof π enables anyone that knows pk to verify the value by $\texttt{VerifyVRF}_{pk}$. In our protocol, $(hash, \pi)$ is always appended to a message and hence not explicitly specified. $\texttt{VerifyVRF}_{pk}(msg, x)$ verifies that $hash$ is the correct value computed from x by using π. The appended $hash$ value is denoted by $msg.vrf$. We omit the notation of sk, pk and the appended $(hash, \pi)$ when the context is clear. In some of the protocols, VRFs may be used to elect leaders and/or committees to obtain player replaceability, i.e., every step of the protocol is executed by a potentially new set of parties. This approach was pioneered in [9] to construct protocols secure under fully adaptive adversaries that are also efficient, i.e., subquadratic communication complexity.

5.3 MAIN RESULTS: A PLAYER REPLACEABLE PROTOCOL

Our main result is the first player replaceable BFT protocol that has strong (maximum) forensic support. In particular, we construct a partially synchronous, player replaceable BFT protocol (§5.3.1) that tolerates $t = (1 - \epsilon)n/3$ Byzantine faults for safety and liveness while providing forensic support with $t < f \leq (1 - \epsilon)2n/3$, where ϵ is a constant and can be any constant in $(0, 1)$. We provide the forensic protocol and formally prove that when there is a safety violation, the protocol can hold at least $\lceil \lambda/3 \rceil$ Byzantine replicas culpable with irrefutable proof, which is the best possible result (§5.3.3). On the other hand, we present a negative result on forensics (§5.3.4) when making HotStuff [2] player replaceable using ideas in Algorand [106], inspired by which, our protocol is equipped with an additional step, called *certified transition*, to obtain both player replaceability and strong forensic support.

5.3.1 Construction of a Player Replaceable BFT Protocol with Strong Forensic Support

Intuition. In round-based protocols like HotStuff, messages of round r may not arrive at some honest replicas due to partial synchrony. These replicas do not update their states according to round-r messages and have "stale" states. For example, in HotStuff, if an honest

party commits, it must have received $2t+1$ commit messages and thus at least $t+1$ honest parties are locked on the committed value. However, up to $n/3$ honest parties may not be locked and have such a stale state. In a non-player-replaceable world, since a majority of honest parties guard the safety of the commit, there does not exist a $(2t+1)$-sized quorum that votes for a different value when $f \leq t$. When $f > t$, it has been shown in [89] that we can detect $t+1$ such parties whenever there is a safety violation. Intuitively, the idea uses a quorum intersection between the $2t+1$ parties that send a commit message for the first committed value and a specific set of $2t+1$ parties that vote on a different value later.

Unfortunately, with player replaceable protocols, such an argument does not apply. Since only a small λ-sized fraction (λ is a security parameter) of parties are chosen each time, it is highly likely that a replica is elected in the committee only once. Thus, Byzantine replicas can deliberately mimic the behavior of honest replicas who suffer long message delays. This makes it difficult to distinguish between Byzantine replicas from honest replicas and thus, as is, we may not have any forensic support under this circumstance when $f > t$ while still being safe and live when $f \leq t$.

To address this concern, the intuition of our protocol is to enforce replicas to wait for enough messages (2/3 of the committee size) to form a *transition certificate (TC)* of each round r before entering round $r+1$. Waiting for messages of round r ensures that a replica's state is up-to-date at the beginning of round $r+1$. Therefore, no honest replicas have stale states and we can distinguish honest replicas who suffer long message delays from Byzantine replicas, and have strong forensic support. Starting from this intuition, we design the protocol with safety and liveness properties as well as strong forensic support.

Protocol overview. The protocol proceeds in a sequence of consecutive rounds where each round lasts for at least 4Δ time. In each round, a set of leaders and a committee will be self-selected from all replicas using cryptographic sortition. The role of a leader is to collect *votes* from committee members and generate a *quorum certificate (QC)* from the votes. It then proposes a *block* that contains the QC to all replicas. The role of a committee member is to wait for the leader's proposal and if it is valid, vote for it. We first describe the sortition process used for election, then define the data structures used in the protocol, and finally present the protocol.

Cryptographic sortition. We use cryptographic sortition to choose a random subset of parties as leaders or committee members, by using VRF [107]. A replica determines its eligibility to be the next leader or the committee member by computing VRF from the random seed, the round, and the role (''*leader*'' or ''*committee*''), i.e., $\texttt{VRF}_{sk}(seed||curRound||role)$, $role \in$

$\{$''*leader*'', ''*committee*''$\}$. If the VRF hash value is smaller than a threshold, the replica is eligible to play the role. When it fulfills its role by broadcasting a message, it accompanies the VRF output (hash value and proof), allowing other replicas to verify its eligibility. For a message m, we denote the accompanied VRF hash value as $m.vrf$. The threshold is set to τ/n for leader and λ/n for committee where τ and λ are the expected number of leaders and the committee size, respectively. Hence, to validate cryptographic sortition of a message m, a replica calls `VerifyVRF` and checks whether $m.vrf < \tau/n$ or λ/n appropriately. To ensure that some block is proposed in each round with high probability, parameter τ should be chosen much larger than 1, e.g., Algorand [21] chooses $\tau = 26$. Let $t_H \leftarrow \lceil 2\lambda/3 \rceil$ where λ is a security parameter. Thus, when $f < (1-\epsilon)n/3$, with high probability, we have more than 2/3 honest replicas in the committee.

Cryptographic sortition enables player replaceability in a straightforward manner. In each round, a new leader and committee are elected privately, i.e., only the elected parties know their eligibility before they fulfill their roles. To be resilient to strongly adaptive adversaries, the protocol can use ephemeral keys as in Algorand [9]. For simplicity, we use the same random seed in the genesis block for cryptographic sortition in all rounds. The protocol can be enhanced with a frequently refreshing random seed, as in Algorand [21]. Cryptographic sortition also works in a proof-of-stake setting if eligibility is weighted by stakes.

Blocks and quorum certificates. Client requests are batched into *blocks*. Each block references its predecessor (parent) with the exception of the genesis block which has no predecessor. A block proposed in round r, denoted b_r, has the following format: $b_r := (cmd, parent, justify)$. *cmd* denotes client commands to be committed, *parent* denotes the hash of the parent block of block b_r, and *justify* stores the quorum certificate (QC) for the parent block. A QC for a block b_r consists of at least t_H vote messages. A QC contains the hash, the round number of the block, and metadata such as signatures and accompanying VRF outputs of the vote messages. Notice that we abuse the notation *qc.block* to refer to the actual block instead of the block hash when the context is clear. A block is said to be valid if its parent is valid (genesis block is always valid) and client requests in the block meet application-level validity conditions. A block b_r extends a block $b_{r'}$ if $b_{r'}$ is an ancestor of b_r. Note that a block extends itself. Two blocks b_r and $b'_{r'}$ are conflicting if they do not extend one another.

5.3.2 Full Protocol and Safety and Liveness

In this section, we provide the full protocols, as well as formal proofs for safety and liveness when $f < (1-\epsilon)n/3$.

Full protocol. Algorithm 5.1 describes the full protocol. Each replica maintains a lock denoted as *lockedQC* initialized as $qc_{genesis}$, and a set TC consisting of a set of locks for every round. Each round proceeds as follows.

- **Propose.** A replica checks its potential leader eligibility using cryptographic sortition (line 5). Leader can construct a new QC and update its own lock once receiving t_H votes for the same block. Then the leader collects commands and proposes a new block extending from *lockedQC.block*.
- **Process proposals.** Unlike HotStuff, a replica waits for a fixed length period (period $[0, 2\Delta)$) for proposals in case there are multiple leaders eligible to propose (line 11). When a replica receives multiple proposals, it chooses the one with the smallest VRF hash (line 13). At time 2Δ of this round, all replicas check the validity of the block and the *safety rule* to ensure the new block extends from *lockedQC.block* (line 15). If the block is valid and safe, replicas will update $lockedQC, TC$ properly. When three consecutive QCs are formed, the block is *directly* finalized, and all its previous blocks on the same chain will also be finalized *indirectly* (line 32).
- **Vote and timeout.** Then every replica checks its eligibility to vote for the round (line 18). The vote message is denoted as $\langle \text{VOTE}, r, b, lockedQC, TC[r-1] \rangle$, where r is the current round number, b the hash of the block the committee replica votes for, and $TC[r-1]$ the set of locks collected from the last round (line 19). When $b = \emptyset$, the vote message serves as a timeout message and meanwhile contains the lock of the committee replica. When b is not empty, it is required that $b.justify = lockedQC$. For each round, replicas selected as committee broadcast their votes to all replicas.
- **Wait for locks.** All replicas cannot enter a round r until they receive t_H locks reported by the committee in round $r-1$. If a vote message in r is received from a replica whose lock has not been received in this round, the lock is added into a set $TC[r]$, and if the lock is more up-to-date, the replica updates its own lock (line 23). The replicas will also update $TC[r-1]$ given $TC^*[r-1]$ contained in the vote. At time 4Δ or later of a round r, replicas enter the next round $r+1$ if $|TC[r]| \geq t_H$.

The safety follows from the following arguments.

Lemma 5.1. For any valid QC, QC', if $QC.round = QC'.round$, $QC.block$ and $QC'.block$ are not conflicting except with $\exp(-\Omega(\lambda))$ probability.

Proof. To show a contradiction, suppose QC, QC' are formed in the same round and $QC.block$, $QC'.block$ are conflicting, then at least $2t_H$ distinct votes are generated by the committee in the same round. Since honest replicas will only vote for one block, we require the following inequality to hold: $|H| + 2|B| \geq 2t_H$, where H and B are the set of honest and Byzan-

Algorithm 5.1 A player replaceable, partially synchronous SMR protocol

1: $t_H \leftarrow \lceil 2\lambda/3 \rceil$
2: $lockedQC \leftarrow qc_{genesis}$ ▷ the lock variable
3: **for** $curRound \leftarrow 1, 2, \ldots$ **do**
▷ **At time 0 of *curRound***
4: $TC[curRound] \leftarrow \emptyset$
5: **if** $\texttt{VRF}_{sk}(seed||curRound||\text{''}leader\text{''}) < \tau/n$ **then** ▷ as the leader of $curRound$
6: **if** the number of votes in $curRound - 1$ for the same block $\geq t_H$ **then**
7: $lockedQC \leftarrow$ QC generated from these Vote messages
8: create block b^* where $b^*.justify \leftarrow lockedQC$; $b^*.cmd \leftarrow$ commands from clients
9: broadcast $\langle \text{Proposal}, curRound, b^*, TC[curRound-1] \rangle$
10: $m \leftarrow \emptyset$
11: wait for $m' \leftarrow \langle \text{Proposal}, curRound, b^*, TC[curRound-1] \rangle$ from a leader whose cryptographic sortition is valid
12: **if** $(m = \emptyset) \vee (m'.vrf < m.vrf)$ **then**
13: $m \leftarrow m'$ ▷ m is the proposal with the min VRF hash
▷ **At time 2Δ of *curRound***
14: $blockHash = \emptyset$
15: **if** m is not empty and $m.b^*$ extends from $lockedQC.block$ **then**
16: ProcessProposal(m) (line 27)
17: $blockHash = H(m.b^*)$
18: **if** $\texttt{VRF}_{sk}(seed||curRound||\text{''}committee\text{''}) < \lambda/n$ **then** ▷ as a committee member of $curRound$
19: broadcast $\langle \text{Vote}, curRound, blockHash, lockedQC, TC[curRound-1] \rangle$
20: **while** $|TC[curRound]| < t_H$ or before 4Δ of $curRound$ **do**
21: wait for $\langle \text{Vote}, curRound, h^*, lockedQC, TC[curRound-1] \rangle$ whose cryptographic sortition is valid
▷ **At any time (triggered by receiving a vote)**
22: **upon receiving** $\langle \text{Vote}, r, h^*, lockedQC^*, TC^*[r-1] \rangle$ s.t. cryptographic sortition is valid **do**
23: **if** sender id has no entry in $TC[r]$ **then**
24: $TC[r] \leftarrow TC[r] \cup \{(id, lockedQC^*)\}$
25: $lockedQC \leftarrow \max_{round}\{lockedQC^*, lockedQC\}$ ▷ do not update in case of a draw
26: $TC[r-1] \leftarrow TC[r-1] \cup TC^*[r-1]$ ▷ do not update in case an entry for an id exists
27: **procedure** ProcessProposal($\langle \text{Proposal}, r, b^*, TC^*[r-1] \rangle$)
28: $lockedQC \leftarrow \max_{round}\{b^*.justify, lockedQC\}$ ▷ do not update in case of a draw
29: $TC[r-1] \leftarrow TC[r-1] \cup TC^*[r-1]$ ▷ do not update in case an entry for an id exists
30: $b' \leftarrow b^*.parent$, $b \leftarrow b'.parent$
31: **if** b, b', b^* are in consecutive rounds **then**
32: finalize block b (directly) and all blocks before b (indirectly), execute commands in the finalized blocks

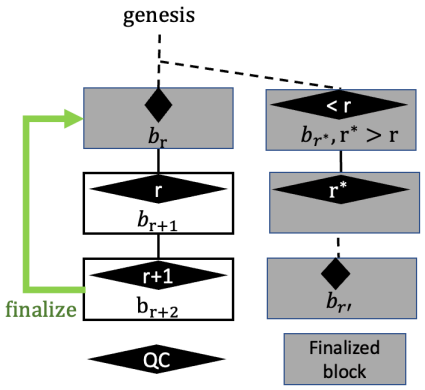


Figure 5.1: A case where two conflicting blocks are both finalized.

tine committee members respectively. By Chernoff bound, the probability of this event is $\exp(-\Omega(\lambda))$ since there are at most $(1-\epsilon)n/3$ Byzantine replicas. QED.

Theorem 5.1. Any two conflicting blocks will not be both finalized by honest replicas except with $\exp(-\Omega(\lambda))$ probability.

Proof. For contradiction, suppose two conflicting blocks are finalized by two honest replicas (see Figure 5.1). Let $b_r, b_{r'}$ be the first directly finalized blocks (finalized in $r+2, r'+2$ by two consecutive QCs) that are conflicting, w.l.o.g., assume $r < r'$. Further, we can assume $r+1 < r'$ due to Lemma 5.1. In round $r+2$, b_r is finalized, at least t_H committee replicas in round $r+1$ receive b_{r+1} and update lock to at least QC_r (the QC containing votes in round r) if they are honest.

Then consider the first block b_{r^*} (possibly $b_{r'}$) conflicting with b_r and proposed after $r+1$. The $b_{r^*}.justify$ must be formed in a round $< r$ because b_{r^*} is the first conflicting block after $r+1$. Since $b_{r'}$ is finalized, a QC for b_{r^*} must be formed, which means at least t_H votes are generated in round r^*.

To enter $r^* > r+1$, replicas need to collect at least t_H locks from committee members in $r+1$. Remember at least t_H committee replicas in round $r+1$ are locked on QC_r, then for every replica, $TC[r+1]$ must contain at least one QC_r reported by honest replica except with $\exp(-\Omega(\lambda))$ probability. Since honest replicas who are locked on QC_r cannot vote for a staler lock and $f < (1-\epsilon)n/3$, it is contradictory that a QC for b_{r^*} is formed. QED.

Since the protocol is synchronous after GST, the liveness satisfies the following statements.

Lemma 5.2. After GST, if an honest replica enters some round r at time T, all honest replicas enter round r by time $T+4\Delta$ except with $\exp(-\Omega(\lambda))$ probability.

Proof. Suppose the earliest honest replica enters round r at time T (after GST). If this honest replica is a committee replica in round r, it will broadcast $TC[r-1]$. Due to synchrony, all

honest replicas will receive $TC[r-1]$ within $T+2\Delta$ time and enter round r. Otherwise, since there are t_H messages in $TC[r-1]$ sent by committee replicas in round $r-1$, at least t_H replicas have entered $r-1$ before $T-\Delta$, among which at least one is honest except for probability $\exp(-\Omega(\lambda))$ since $f < (1-\epsilon)n/3$. The honest committee replica in round $r-1$ will broadcast $TC[r-2]$, therefore at T, all honest replicas will enter round $r-1$. Then honest committee replicas in $r-1$ will broadcast votes before $T+2\Delta$. And since there are at least t_H honest committee replicas except with $\exp(-\Omega(\lambda))$ probability, all honest replicas can collect $TC[r-1]$ by $T+4\Delta$. QED.

Lemma 5.3. At any point after GST, if the highest QC is QC_r, and three consecutive rounds $r+1 \sim r+3$ have honest leaders, then b_{r+1} will be finalized within 16Δ time except with $\exp(-\Omega(\lambda))$ probability.

Proof. After GST, when QC_r is sent by leader $r+1$, by Lemma 5.2, all honest replicas enter $r+1$ within 4Δ. If the leaders of $r+1 \sim r+3$ are honest, leader $r+1$ will propose b_{r+1} extending from the parent of b_r and all honest replicas are willing to vote. The leader $r+2$ can collect QC_{r+1} except with $\exp(-\Omega(\lambda))$ probability and propose b_{r+2}. Similarly leaders $r+3$ can collect enough votes to form QC_{r+2}. Then b_{r+1} is finalized within 3 rounds, each round costs at most 4Δ. QED.

Theorem 5.2 (Liveness)**.** All honest replicas keep finalizing new blocks and valid transactions will be finalized.

Proof. Since leader changes in every round randomly, the probability to elect consecutive 3 honest leaders is $> (2/3)^3$. Whenever a QC is formed and 4 honest leaders are elected consecutively, by Lemma 5.3, a block will be finalized within a time bound with overwhelming probability. Thus all honest replicas will keep finalizing new blocks. Honest leaders will add valid transactions into their proposals, hence valid transactions will be finalized. QED.

5.3.3 Forensic Protocol and Proof of Forensic Support

When $f < (1-\epsilon)n/3$, the safety and liveness of Algorithm 5.1 are proved as above. When $f \geq (1-\epsilon)n/3$, it is possible that the safety is violated. In such a case, the following forensic protocol in Algorithm 5.2 can provide forensic support proved in Theorem 5.3.

Theorem 5.3. When $f \geq (1-\epsilon)n/3$, if two honest replicas finalize conflicting blocks, the protocol in Algorithm 5.1 provides $((1-\epsilon)2n/3, 2, \lceil \lambda/3 \rceil)$-forensic support.

Proof. Suppose two conflicting blocks are finalized by two honest replicas, let $b_r, b_{r'}$ be the first directly finalized blocks that are conflicting, w.l.o.g., suppose $r \leq r'$.

Algorithm 5.2 Forensic protocol for Algorithm 5.1

1: **upon receiving** conflicting blocks finalized by two honest replicas **do**
2: query the entire blockchain from the two honest replicas
3: find the first block finalized by consecutive QCs in each chain, denoted by $b_r, b_{r'}$
4: swap $b_r, b_{r'}$ if $r' < r$ ▷ make sure $r \leq r'$
5: **if** $r + 2 > r'$ **then**
6: find two $QC_{r'}$ on each chain
7: return the intersection of two $QC_{r'}$
8: **else**
9: query $TC[r + 1]$ from either of the honest replicas
10: **if** all *lockedQC* in $TC[r + 1]$ has round $< r$ **then**
11: find QC_{r+1} that makes b_r be committed
12: return the intersection of $TC[r + 1]$ and QC_{r+1}
13: **else**
14: find block b_{r^*} s.t.
(1) $r + 2 \leq r^* \leq r'$, and
(2) $b_{r'}$ extends b_{r^*}, and
(3) b_r conflicts with b_{r^*}, and
(4) r^* is the smallest round satisfying the above 3 conditions
15: find QC for b_{r^*}, denoted by QC_{r^*}, return all replicas in QC_{r^*}

Case $r + 2 > r'$. Culpability. If $r \leq r' < r + 2$, there are two quorums formed in r', these two $QC_{r'}$ intersect in $\lceil \lambda/3 \rceil$ replicas. These replicas should be Byzantine since the protocol requires a replica to vote for at most one block in a round.
Witnesses. In this case, the culpability proof can be constructed from two QCs generated in the same round (line 5-7, Algorithm 5.2).

Case $r+2 \leq r'$. Culpability. Since b_r is directly finalized in round $r+2$ (by QC_{r+1}), it must be the case that at least t_H committee replicas are locked on at least QC_r after round $r + 1$ (if they are honest), and broadcast their vote with lock to all replicas. Then consider the first block b_{r^*} (possibly $b_{r'}$) that is conflicting with b_r and proposed after $r + 1$. On the one hand, b_{r^*} must be extended from a block older than b_r since this is the first conflicting block proposed after r. On the other hand, only those replicas whose locks are staler than QC_r can vote for b_{r*}. Remember that in round $r + 1$, at least t_H committee replicas broadcast lock QC_r (or higher lock). And for committee replicas in r^* to vote, they must collect a set $TC[\cdot]$ consisting of at least t_H locks from the committee in every round $< r^*$. If the lock of any one of them is still staler than QC_r, the intersection ($\lceil \lambda/3 \rceil$ replicas) of QC_{r+1} and $TC[r + 1]$ is the set of committee replicas who send incompatible locks hence are Byzantine (line 10-12, Algorithm 5.2). Otherwise all the committee replicas who vote for b_{r*} must be Byzantine (line 13-15, Algorithm 5.2).

Witnesses. In this case, there are two possible scenarios. (i) QC_{r+1} intersects $TC[r+1]$ in $\lceil \lambda/3 \rceil$ replicas, who are culpable since their votes in QC_{r+1} and $TC[r+1]$ are incompatible. (ii) All replicas in QC_{r^*} (at least t_H in total) are Byzantine because they should have received a $TC[\cdot]$ containing QC_r and update their locks to be at least QC_r, but they vote for a conflicting block b_{r^*} extending from a block older than r. These two cases indicate that with same-round safety violation, the witnesses can detect $\lceil \lambda/3 \rceil$ replicas. If same-round safety violation does not exist, at least t_H culprits can be detected.

QED.

5.3.4 Impossibility for Forensic Support in Absence of Transition Certificate

The previous section shows a player replaceable protocol that has strong forensic support. In this section, we will show that not all player replaceable protocols have this property even if they are safe and live. In fact, we will consider a natural protocol in the same vein as was discussed in the previous section and demonstrate an impossibility of forensic support when transition certificates are not forwarded.

The protocol is described in Algorithm 5.3 as a diff from Algorithm 5.1. The key differences are: (1) Leaders and committee members do not need to maintain TC of the last round in their messages. (2) Replicas do not need to wait until collecting enough locks reported by committee to enter the next round. (3) The committee only sends votes when a valid proposal is received.

Intuition for the impossibility. The protocol is modified from two-phase HotStuff [2] (or Tendermint [16]) with player replaceability. Thus, safety and liveness can be guaranteed in a similar way with high probability. In terms of forensic support, when safety is violated within a round, the culprits can still be detected from the intersection of two conflicting QCs formed in the same round. However, the absence of the TC results in the impossibility of detection when safety violation happens across rounds, since the behavior of committee members in one round is not traceable to another round. When $f \geq (1-\epsilon)n/3$, suppose safety is violated such that two conflicting blocks are finalized by two honest replicas. Let $b_r, b_{r'}$ be the first directly finalized blocks that are conflicting, w.l.o.g., suppose $r < r'$. When $r+1 < r'$, consider the first block b_{r^*} on the same blockchain as $b_{r'}$ (possibly $b_{r^*} = b_{r'}$) that is proposed after $r+1$, b_{r^*} must extend from a block older than b_r. Since $b_{r'}$ is finalized, QC_{r^*} must be formed. Note that QC_{r+1} contains at least t_H committee replicas who are locked on at least QC_r (if they are honest); QC_{r^*} contains at least t_H committee replicas who vote for b_{r^*} which extends from an older block than b_r. Denote $s = QC_{r+1} \cap QC_{r^*}$,

Algorithm 5.3 A player replaceable, partially synchronous SMR protocol without TC

1: $t_H \leftarrow \lceil 2\lambda/3 \rceil$
2: $lockedQC \leftarrow qc_{genesis}$ ▷ the lock variable
3: **for** $r = 1, 2, \ldots$ **initialize**:
4: ~~$TC[r] \leftarrow \emptyset$~~
5: **for** $curRound \leftarrow 1, 2, \ldots$ **do**
▷ **At time 0 of *curRound***
6: **if** $\texttt{VRF}_{sk}(seed||curRound||\text{''}leader\text{''}) < \tau/n$ **then** ▷ as the leader of $curRound$
7: **if** the number of votes ~~in $curRound - 1$~~ for the same block $\geq t_H$ **then**
8: $lockedQC \leftarrow$ QC generated from these Vote messages
9: create block b^* where $b^*.justify \leftarrow lockedQC$; $b^*.cmd \leftarrow$ commands from clients
10: broadcast $\langle \text{PROPOSAL}, curRound, b^* \rangle$
11: $m \leftarrow \emptyset$
12: wait for $m' \leftarrow \langle \text{PROPOSAL}, curRound, b^* \rangle$ from a leader whose cryptographic sortition is valid
13: **if** $(m = \emptyset) \vee (m'.vrf < m.vrf)$ **then**
14: $m \leftarrow m'$ ▷ m is the proposal with the min VRF hash
▷ **At time 2Δ of *curRound***
15: $blockHash = \emptyset$
16: **if** m is not empty and $m.b^*$ extends from $lockedQC.block$ **then**
17: ProcessProposal(m) (line 27)
18: $blockHash = H(m.b^*)$
19: **if** $\texttt{VRF}_{sk}(seed||curRound||\text{''}committee\text{''}) < \lambda/n$ **then** ▷ as a committee member of $curRound$
20: broadcast $\langle \text{Vote}, curRound, blockHash \rangle$
21: **while** ~~$|TC[curRound]| < t_H$~~ or before 4Δ of $curRound$ **do**
22: wait for $\langle \text{Vote}, curRound, h^* \rangle$ whose cryptographic sortition is valid
▷ **At any time (triggered by receiving a vote)**
23: **upon receiving** $\langle \text{Vote}, r, h^*, lockedQC^* \rangle$ s.t. cryptographic sortition is valid **do**
24: **if** sender id ~~has no entry in $TC[r]$~~ **then**
25: ~~$TC[r] \leftarrow TC[r] \cup \{(id, lockedQC^*)\}$~~
26: $lockedQC \leftarrow \max_{round}\{lockedQC^*, lockedQC\}$ ▷ do not update in case of a draw
27: ~~$TC[r-1] \leftarrow TC[r-1] \cup TC^*[r-1]$~~
28: **procedure** ProcessProposal($\langle \text{PROPOSAL}, r, b^* \rangle$)
29: $lockedQC \leftarrow \max_{round}\{b^*.justify, lockedQC\}$ ▷ do not update in case of a draw
30: ~~$TC[r-1] \leftarrow TC[r-1] \cup TC^*[r-1]$~~
31: $b' \leftarrow b^*.parent$, $b \leftarrow b'.parent$
32: **if** b, b', b^* are in consecutive rounds **then**
33: finalize block b (directly) and all blocks before b (indirectly), execute commands in the finalized blocks

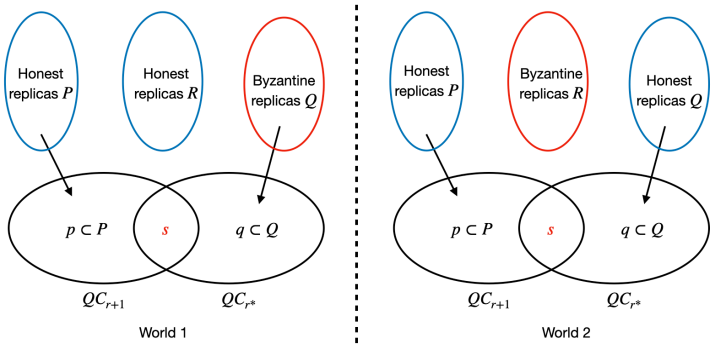


Figure 5.2: Two worlds where both QC_{r+1} and QC_{r^*} are generated for conflicting blocks.

replicas in s should be held culpable since they violate the voting rules. However, when n is large, due to player replaceability, in most cases s is empty. The following arguments show the impossibility of forensic support by analyzing the size of s.

Lemma 5.4. Let $b_r, b_{r'}$ be the first directly finalized blocks that are conflicting, where $r+1 < r'$. Let b_{r^*} be the first block on the same blockchain as $b_{r'}$ (possibly $b_{r^*} = b_{r'}$) that is proposed after $r+1$. Let $s = QC_{r+1} \cap QC_{r^*}$ be the intersection of two QCs. At most $|s|$ replicas can be held culpable with irrefutable evidence.

Proof. Consider the executions in rounds $r+1$ and r^*, where a QC_{r+1} is collected by leader in $r+2$ and a conflicting QC_{r^*} is formed where $r^* > r+1$. Let $s = QC_{r+1} \cap QC_{r^*}$, $p = QC_{r+1}/s$ and $q = QC_{r^*}/s$. Let there be four replica partitions P, Q, R and s, s.t. $p \subset P, q \subset Q$, $|Q| = |R| \geq (1-\epsilon)n/3 - |s|$ and $|P| = n - |Q| - |R| - |s|$. We present the following two worlds with different sets of replicas being Byzantine. Notice that when $f \geq (1-\epsilon)n/3$, it is possible that the fraction of Byzantine replicas in committee exceeds $1/3$. With larger f, the probability becomes higher.

World 1: Let Q, s be Byzantine replicas in this world. In round $r+1$, P, s receive b_{r+1} and update their locks to b_r if they are honest. The votes from p, s forms QC_{r+1}. Later in r^*, a leader in R proposes a conflicting block b_{r^*} since it did not receive block b_{r+1}. Now Q, s receive the proposal and the votes from q, s forms QC_{r^*}. The intersection of QC_{r+1} and QC_{r^*} is s.

World 2: Let R, s be Byzantine replicas in this world. In round $r+1$, P, s receive b_{r+1} and update their locks to b_r if they are honest. The votes from p, s forms QC_{r+1}. Later in r^*, a Byzantine leader in R proposes a conflicting block b_{r^*}, now Q, s receive the proposal. Since they did not receive b_{r+1}, their locks are staler than b_r. Once the parent block of b_{r^*} is not older than the locks of q, they will vote for b_{r^*}. The votes from q, s forms QC_{r^*}.

Even with transcripts from all honest replicas, based on QC_{r+1} and QC_{r^*}, World 1 and World 2 are indistinguishable to everyone other than parties in Q and R. The only distinction is whether these parties received QC_{r+1}. The culpability of s is easy to prove. However, if the protocol can provide an irrefutable proof of $d > |s|$ culprits, someone in Q or R will be mistakenly blamed in at least one of the above worlds. QED.

Lemma 5.4 demonstrates that only the replicas in the intersection of two QCs will be held culpable. Any Byzantine replicas that only vote for one QC can not be distinguished from honest replicas that hold stale locks since there is no evidence that they have a more updated lock with a conflicting block. In the following theorem, we further prove that this number is possibly zero due to player replaceability.

Theorem 5.4. With n replicas, when $f \geq (1-\epsilon)n/3$, if two honest replicas finalize two conflicting blocks, the protocol has no forensic support ($d = 0$) with non-negligible probability.

Proof. No matter how large f is, we assume there are $f^* = (1-\epsilon)n/3$ Byzantine replicas plan to equivocate in two rounds. Because if $f^* < (1-\epsilon)n/3$, the probability to launch an attack is $\exp(-\Omega(\lambda))$, which is negligible. And if $f^* \geq (1-\epsilon)n/3$, larger f^* can increase the probability of attack, but will also increase the expected number of Byzantine replicas that will be detected. So Byzantine replicas can always choose to let f^* replicas equivocate.

Now denote X to be the number of replicas who both vote for QC_{r+1} and QC_{r^*}. Then we have

$$\Pr[X = s] = \binom{(1-\epsilon)n/3}{s} \left(\frac{\lambda}{n}\right)^{2s} \cdot \left(1 - \left(\frac{\lambda}{n}\right)^2\right)^{((1-\epsilon)n/3-s)} \tag{5.1}$$

Denote the cumulative distribution function as

$$F(s; (1-\epsilon)n/3, (\lambda/n)^2) = \Pr[X \leq s] \tag{5.2}$$

the probability to detect at least $d > 0$ culprits is $1 - F(d-1; (1-\epsilon)n/3, (\lambda/n)^2)$. Specially, the probability that there is no replica in the intersection is $\Pr[X = 0] \sim \exp(-(\lambda^2/n))$, which can be non-negligible (e.g., $\lambda = O(\sqrt{n \log n})$).

QED.

The need for transition certificate. Notice that the only difference between Algorithms 5.1 and 5.3 is the use of transition certificates. Intuitively, the key point of forensic protocol is to detect contradicting behaviors of Byzantine replicas in non-trivial cases (across rounds), but under the player replaceable setting, the continuity of participation of single

replica is lost. Algorithm 5.1 asks all replicas in the committee broadcast TC to announce how they enter the new round, and the TC connects rounds and committees even if the committees in different rounds are mutual exclusive.

5.4 AN EXTENSION: RECONFIGURABLE PROTOCOLS

Unlike player replaceable protocols that use VRFs to select players randomly and secretly in each step, many practical blockchain protocols choose to substitute players in a more straightforward and flexible way. Protocol designers can appoint a set of players for each epoch and stipulate a fixed number of rounds/views for an epoch, e.g., 100. In this way, the protocol changes players in every 100 rounds based on the designer's appointment. The designer can also specify a selection based on stakes (e.g., top 10 stakeholders) or performance (e.g., top 10 network communication contributors). The protocols with the relaxed player selection, also called reconfigurable protocols, face the same difficulty in providing forensic support. For instance, DiemBFT [10] adopts a protocol similar to HotStuff and plans to reconfigure players in epoch switches. Although HotStuff provides strong forensic support, this property only holds within an epoch, and there is no guarantee of forensic support during epoch switches. Fortunately, the notion of transition certificates can be adapted to these protocols and provide strong forensic support during epoch switches. We provide a framework with requirements for reconfigurable protocols and instruction to provide strong forensic support.

5.4.1 Forensic Framework

Intuition. When an epoch switch happens, the old set of players is substituted with a new set of players, and these two sets likely have no intersection. Thus, the forensic analysis based on quorum intersection does not apply, and Byzantine parties can deliberately mimic the behavior of honest replicas who suffer long message delays. Byzantine behavior is indistinguishable from honest behavior, and we may not have any forensic support. This intuition is similar to §5.3.

Protocol requirements. The protocol should be partially synchronous and tolerate a maximum of $(1 - \epsilon)/3$ of Byzantine adversaries. It should run in consecutive rounds (or views) that are counted by a variable *curRound*, and no round should be skipped. Each player should maintain protocol's local states that affect the correct execution and store them collectively as $State_{curRound}$, where the underscript of *curRound* indicates this state is at the

end of round *curRound*. If a player misses or corrupts its local $State_{curRound}$, it will deviate from the protocol and perform Byzantine behavior after *curRound*. As for communication, one player can send messages directly to others or broadcast messages. Messages are digitally signed by the sender, and the private key of the signature should be kept secret. The protocol should require all the related messages to be received before finalization (e.g., receiving the entire blockchain that contains QC, as in chained HotStuff/DiemBFT).

Epochs are delimited by rounds. For any round r, public indicator IsEpochStart(r) and IsEpochEnd(r) should tell if the round is a start or end of an epoch. Notice that the first epoch start (round 1) is special and we stipulate IsEpochStart(1) = False. Public function Players(r) should return the list of players of round r which has size λ. Also, this function should have the same list within an epoch. The list of players should also contain $> 2\lambda/3$ honest players under normal adversary. Notice that if we consider the adversary that can create a safety violation, the list of players may contain $\leq \lambda/3$ honest players. Round r of the protocol is executed by players in Players(r), and if the next round $r+1$ is a start of a new epoch and Players($r+1$) is a new set of players, an epoch switch and a reconfiguration happen. The protocol should have strong forensic support by using the quorum intersection technique within an epoch as the protocols analyzed in Chapter 4. If a safety attack happens during an epoch switch, the forensic protocol should return two quorums (but unable to intersect them).

Forensic framework. We denote the original reconfigurable protocol by Π. Algorithm 5.4 shows the forensic framework of Π, and the framework proceeds in rounds. Besides executing Π, we add the following procedures in each round:

1. If it is the end of an epoch, the old set of players should send local state *State* to the new set of players (line 10). This procedure helps the new set of players collect the correct states; thus, they can run the new epoch without deviating from the protocol.
2. If it is the start of an epoch, the new set of players should collect the state messages from the old set of players (line 3). The new set of players updates their state according to the old ones (line 4). This procedure helps the new set of players update their state to the correct one.
3. If it is the start of an epoch, the new set of players should wait until it collects 2/3 of the old players' state (line 5). This procedure forces the new players to continue until they have the updated state.
4. The new set of players should append the collected states as the transition certificate to their messages (line 6). (Signature is of the whole message, including the transition certificate.) All nodes, including players and those who only listen to messages, validate

messages about the transition certificate (line 7). This procedure enforces the execution of the previous procedure.

Algorithm 5.4 A forensic framework for reconfigurable protocol Π

1: **for** $curRound \leftarrow 1, 2, \ldots$ **do**
2: **if** IsEpochStart($curRound$) and it belongs to Players($curRound$) **then**
3: collect $State_{curRound-1}$ from Players($curRound - 1$) and store them in $TC[curRound]$
4: update local $State$ according to received $State_{curRound-1}$
5: wait until $|TC[curRound]| \geq t_H$, then continue to the next line
6: when execute Π, append $TC[curRound]$ to messages of all rounds in this epoch
7: when execute Π, validate messages for the appended $TC[r]$ (r should be the start of this epoch): if there are not t_H signed $State_{r-1}$ from Players($r - 1$), discard it
8: execute $curRound$ of Π
9: **if** IsEpochEnd($curRound$) and it belongs to Players($curRound$) **then**
10: send $State_{curRound}$ to players in Players($curRound + 1$)

Safety and Liveness. The framework does not change the finalization part of the protocol. Hence, the safety is untouched. We prove the liveness of the framework.

Theorem 5.5. All honest replicas keep finalizing new blocks and valid transactions will be finalized.

Proof. The framework only takes time negligible to message delay except that it blocks the execution in "waiting for enough states to form a transition certificate" (line 5). For this operation, we need to prove it does not block forever. We prove it using induction on round number r. Before the waiting, this operation is not executed, and it will not block. Consider if it blocks at round r, since the original protocol Π has liveness and due to the induction hypothesis, all players should eventually enter round $r - 1$, so old set of players should send their states, and honest ones' states will arrive at the new set of players. These messages must be no less than t_H. Hence, this operation will not block after these messages arrive. QED.

5.4.2 Forensic Protocol and Proof of Forensic Support

When $f \geq (1 - \epsilon)n/3$, it is possible that safety is violated. For this framework, we only care about the safety violation in an epoch switch, i.e., a part of the adversaries belongs to the old set of players while another part belongs to the new set. The forensic protocol is

Algorithm 5.5 Forensic protocol for Algorithm 5.4

1: **upon receiving** conflicting blocks finalized by two honest replicas **do**
2: find the two quorum Q, Q' (ordered by round) by Π's forensic protocol
▷ **Only care about Q, Q' across two epochs**
3: denote the epoch of Q as the old epoch and that of Q' as the new epoch. Let the new epoch starts from round r_l
4: query the messages of rounds between that of Q, Q', from either of the honest replicas
5: specifically, query $TC[r_l]$, which is appended to messages by the new players
6: **for** player $p \in Q'$ **do**
7: find $TC[r_l]$ sent by p, denoted as tc
8: **if** any of the t_H states inside tc are compatible with the messages related to Q **then**
9: add p to culprits ▷ tc indicates that p should not do as Q' does
10: **else**
11: find the intersection of the incompatible states inside tc and Q
12: add them to culprits ▷ the old players that send incompatible states are malicious

presented in Algorithm 5.5. We can get two quorums by the original protocol Π's forensic protocol, yet we cannot do quorum intersection since their players are reconfigured. We start with the latter quorum and check for all its members whether the transition certificate sent by it is compatible with previous messages. If the transition certificate is compatible, then this member must be malicious. Otherwise, there must be players who sent an incompatible state, and they must be malicious. We have the formal theorem.

Theorem 5.6. When $f \geq (1-\epsilon)n/3$, if two honest replicas finalize conflicting blocks, the framework in Algorithm 5.4 provides $((1-\epsilon)2n/3, 2, \lceil \lambda/3 \rceil)$-forensic support.

Proof. The forensic protocol of Π should return two quorums Q, Q' when two honest replicas finalize conflicting blocks. If they are within an epoch, we are finished with the forensic protocol of Π. Otherwise, Algorithm 5.5 checks all members in Q'. If it enters line 8, it will find the member deviated. There are two reasons: (1) The member's transition certificate says it should have the correct state at the epoch start. (2) From the epoch start to the round of this quorum Q', the member has no reason to corrupt its state (change its state to an incompatible one), otherwise the forensic protocol Π should have returned an earlier quorum. If it enters line 12, then all the states in the transition certificate are incompatible with the first finalized block, which means all players in the intersection of these incompatible state senders and Q are malicious. This intersection itself has size $\lceil \lambda/3 \rceil$. Also, notice that we query messages from two honest replicas. Hence, we have $((1-\epsilon)2n/3, 2, \lceil \lambda/3 \rceil)$-forensic support. QED.

5.5 RELATED WORK

Forensic support. The idea of holding misbehaving participants accountable has been discussed in earlier works [94, 95] for distributed systems in general. For SMR and blockchain, recent works [46, 97, 98] have discussed finality and accountability and designed their consensus protocols with the focus on accountability. Chapter 4 and a recent work [89] formally define forensic support for Byzantine Agreement (BA) and analyze it for protocols such as PBFT [7, 8], HotStuff [2], VABA [3], and Algorand [9, 21]. They show that except Algorand, the other protocols have forensic support depending on their implementation details. Another work [120] introduces the notion of accountable-safety that combines the traditional safety with the ability to hold Byzantine parties accountable, and shows that there exists a trade-off between accountable-safety and liveness. In reference [99], they propose the class of snap-and-chat protocols, which combines a longest chain protocol with a BFT protocol to provide both availability and finality, and in reference [119] they showe that if the BFT protocol provides accountable-safety, then it is inherited by the snap-and-chat protocol.

5.6 DISCUSSION

We begin with two observations about the forensic properties for player replaceable protocols.

First, compared to protocols analyzed in the previous chapter, player replaceable protocols require fewer replicas to send messages; correspondingly, only fewer replicas ($O(\lambda)$) can be held culpable when there is a safety violation even if the total number of Byzantine replicas is far larger ($O(n)$). Whenever forensic support is available, the number of culpable replicas ($d = \lambda/3$ in Algorithm 5.1) is in the same proportion to the quorum size as in the non player replaceable setting. Moreover, this number is independent of f. When there is no forensic support, no replica may be held culpable.

Second, qualitatively, the key difficulty with holding replicas culpable is related to potentially having a different set of replicas participating in each round. In BFT protocols, voting rules stipulate how previous actions impose restrictions on current behavior. Due to player replaceability, voters' behaviors across rounds are less traceable, which can be utilized by adversary to conceal evidence of deviation. Thus, to construct a protocol with strong forensic support, we need to reconnect across-rounds actions of replicas. In our protocol, transition certificates serve as the link between the rounds of ancestor blocks and the current blocks which the forensic protocol can use to identify culpable behavior.

Many practical blockchain systems use reconfigurable protocols in that they want to sub-

stitute players infrequently and need more straightforward ways for choosing players. For these systems and their protocols, our proposal of transaction certificates helps to build a framework to ensure strong forensic support during the reconfiguration. However, the proposed framework has strict requirements on protocols; they must proceed in consecutive rounds without skipping rounds and work with a fixed-size committee. Investigating the feasibility of implementing this framework on top of extant blockchain systems and the necessity of the requirements to obtain the desired forensic property will be a good direction for future work.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[3] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 337–346.

[4] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," *IACR Cryptology ePrint Archive*, vol. 2019, p. 270, 2019.

[5] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[6] V. Bagaria, S. Kannan, D. Tse, G. Fanti, and P. Viswanath, "Prism: Deconstructing the blockchain to approach physical limits," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19, 2019, p. 585–602.

[7] M. Castro, B. Liskov et al., "Practical byzantine fault tolerance," in *OSDI*, vol. 99, 1999, pp. 173–186.

[8] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[9] J. Chen and S. Micali, "Algorand: A secure and efficient distributed ledger," *Theoretical Computer Science*, vol. 777, pp. 155–183, 2019.

[10] T. D. Team, "State machine replication in the diem blockchain," https://developers.diem.com/docs/technical-papers/state-machine-replication-paper/, 2021. [Online]. Available: https://developers.diem.com/docs/technical-papers/state-machine-replication-paper/

[11] "Forensic module for diem," https://github.com/wgr523/libra, 2020. [Online]. Available: https://github.com/wgr523/libra

[12] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, p. 228–234, apr 1980.

[13] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, p. 299–319, dec 1990.

[14] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[16] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus," *arXiv preprint arXiv:1807.04938*, 2018.

[17] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols," in *Proceedings of the second annual ACM symposium on Principles of distributed computing*, 1983, pp. 27–30.

[18] M. O. Rabin, "Randomized byzantine generals," in *24th annual symposium on foundations of computer science (sfcs 1983)*. IEEE, 1983, pp. 403–409.

[19] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[20] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Brief announcement: Byzantine agreement, broadcast and state machine replication with near-optimal good-case latency," in *34th International Symposium on Distributed Computing, DISC*, 2020.

[21] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 51–68.

[22] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-ng: A scalable blockchain protocol," in *13th USENIX symposium on networked systems design and implementation (NSDI 16)*, 2016, pp. 45–59.

[23] Y. Sompolinsky and A. Zohar, "Secure high-rate transaction processing in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2015, pp. 507–527.

[24] H. Yu, I. Nikolic, R. Hou, and P. Saxena, "Ohie: Blockchain scaling made simple," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, may 2020, pp. 112–127.

[25] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, W. Xu, F. Long, and A. C.-C. Yao, "A decentralized blockchain with high throughput and fast confirmation," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 515–528.

[26] A. Kiayias, A. Russell, B. David, and R. Oliynykov, "Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual International Cryptology Conference*. Springer, 2017, pp. 357–388.

[27] B. David, P. Gaži, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 66–98.

[28] P. Daian, R. Pass, and E. Shi, "Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 23–41.

[29] V. Buterin, "Ethereum whitepaper," https://ethereum.org/en/whitepaper/, 2013.

[30] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich et al., "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, 2018, pp. 1–15.

[31] G. Fanti, L. Kogan, S. Oh, K. Ruan, P. Viswanath, and G. Wang, "Compounding of wealth in proof-of-stake cryptocurrencies," in *International Conference on Financial Cryptography and Data Security*, 2019, pp. 42–61.

[32] "Bitcoin energy consumption index," 2018, https://digiconomist.net/BITCOIN-ENERGY-CONSUMPTION.

[33] N. L. Johnson and S. Kotz, *Urn models and their application: an approach to modern discrete probability theory*. Wiley New York, 1977, vol. 77.

[34] H. Mahmoud, *Pólya urn models*. Chapman and Hall/CRC, 2008.

[35] G. Rammeloo, "The economics of the proof of stake consensus algorithm," Medium, 2017, https://medium.com/@gertrammeloo/the-economics-of-the-proof-of-stake-consensus-algorithm-e28adf63e9db.

[36] moh_man, "How does pos stake concept deal with rich becoming richer issue?" Reddit, 2017, https://www.reddit.com/r/ethereum/comments/6x0xv8/how_does_pos_stake_concept_deal_with_rich/.

[37] Trustnodes.com, ""proof of work is the rich get richer squared" says vitalik buterin," Trustnodes, 2018, https://www.trustnodes.com/2018/07/10/proof-work-rich-get-richer-squared-says-vitalik-buterin.

[38] P. Gaži, A. Kiayias, and A. Russell, "Stake-bleeding attacks on proof-of-stake blockchains," in *2018 Crypto Valley Conference on Blockchain Technology (CVCBT)*. IEEE, 2018, pp. 85–92.

[39] O. Schrijvers, J. Bonneau, D. Boneh, and T. Roughgarden, "Incentive compatibility of bitcoin mining pool reward functions," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 477–498.

[40] I. Eyal and E. G. Sirer, "Majority is not enough: Bitcoin mining is vulnerable," *Communications of the ACM*, vol. 61, no. 7, pp. 95–102, 2018.

[41] L. Brünjes, A. Kiayias, E. Koutsoupias, and A.-P. Stouka, "Reward sharing schemes for stake pools," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 256–275.

[42] R. Pass and E. Shi, "Fruitchains: A fair blockchain," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*. ACM, 2017, pp. 315–324.

[43] J. Earls, "The missing explanation of proof of stake version 3," 2017, http://earlz.net/view/2017/07/27/1904/the-missing-explanation-of-proof-of-stake-version.

[44] I. Bentov, R. Pass, and E. Shi, "Snow white: Provably secure proofs of stake." *IACR Cryptology ePrint Archive*, vol. 2016, p. 919, 2016.

[45] E. Wiki, "Proof of stake faqs," https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQs.

[46] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[47] "Controlled supply," bitcoinwiki, 2018, https://en.bitcoin.it/wiki/Controlled_supply\#cite_note-2.

[48] "Mining," Ethereum Wiki, 2018, https://github.com/ethereum/wiki/wiki/Mining.

[49] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification," Technical report, 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep., 2016.

[50] E. Duffield and D. Diaz, "Dash: A privacycentric cryptocurrency," *Self-published*, 2015.

[51] I. Kaiser, "A decentralized private marketplace: Draft 0.1."

[52] R. Pemantle, "A time-dependent version of pólya's urn," *Journal of Theoretical Probability*, vol. 3, no. 4, pp. 627–637, 1990.

[53] B. Bambrough, "A bitcoin halvening is two years away – here's what'll happen to the bitcoin price," *Forbes*, May 2018.

[54] J. Taylor-Copeland, *Coffey vs. Ripple class action complaint*, 2018.

[55] A. Sapirshtein, Y. Sompolinsky, and A. Zohar, "Optimal selfish mining strategies in bitcoin," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 515–532.

[56] K. Nayak, S. Kumar, A. Miller, and E. Shi, "Stubborn mining: Generalizing selfish mining and combining with an eclipse attack," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pp. 305–320.

[57] G. Fanti, L. Kogan, S. Oh, K. Ruan, P. Viswanath, and G. Wang, "Compounding of wealth in proof-of-stake cryptocurrencies," *arXiv preprint arXiv:1809.07468*, 2018.

[58] G. Wang, S. Wang, V. Bagaria, D. Tse, and P. Viswanath, "Prism removes consensus bottleneck for smart contracts," in *2020 Crypto Valley Conference on Blockchain Technology (CVCBT)*, 2020, pp. 68–77.

[59] *Optimism*. [Online]. Available: https://optimism.io/

[60] *ZK-Rollups*. [Online]. Available: https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups/

[61] H. Kalodner, S. Goldfeder, X. Chen, S. M. Weinberg, and E. W. Felten, "Arbitrum: Scalable, private smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 1353–1370.

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

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

[64] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. R. Rain, S. Sezer et al., "Move: A language with programmable resources," 2019.

[65] *My Sql*. [Online]. Available: https://www.mysql.com/

[66] *PostGres*. [Online]. Available: https://www.postgresql.org/

[67] T. Dickerson, P. Gazzillo, M. Herlihy, and E. Koskinen, "Adding concurrency to smart contracts," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2017, pp. 303–312.

[68] P. S. Anjana, S. Kumari, S. Peri, S. Rathor, and A. Somani, "An efficient framework for optimistic concurrent execution of smart contracts," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 83–92.

[69] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," in *International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2019, May 6-7, 2019, Paris, France*, ser. OASIcs, vol. 71, 2019.

[70] S. Pang, X. Qi, Z. Zhang, C. Jin, and A. Zhou, "Concurrency protocol aiming at high performance of execution and replay for smart contracts," *arXiv preprint arXiv:1905.07169*, 2019.

[71] M. Bartoletti, L. Galletta, and M. Murgia, "A true concurrent model of smart contracts executions," in *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020*, vol. 12134. Springer, 2020, pp. 243–260.

[72] *Libra*, accessed February 16, 2020. [Online]. Available: https://github.com/libra/libra

[73] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ser. PODC '19, 2019, p. 347–356.

[74] J. Poon and V. Buterin, "Plasma: Scalable autonomous smart contracts," *White paper*, pp. 1–47, 2017.

[75] *Ethereum 2.0 Specifications*. [Online]. Available: https://github.com/ethereum/eth2.0-specs

[76] *NEAR Protocol | A sharded, developer-friendly, proof-of-stake public blockchain*. [Online]. Available: https://nearprotocol.com/

[77] *Polkadot: Decentralized Web 3.0 Blockchain Interoperability Platform*. [Online]. Available: https://polkadot.network/

[78] *RocksDB*. [Online]. Available: https://rocksdb.org/

[79] *rust-rocksdb*, accessed February 19, 2020. [Online]. Available: https://github.com/rust-rocksdb/rust-rocksdb

[80] *OpenEthereum*, accessed January 30, 2020. [Online]. Available: https://github.com/openethereum/openethereum

[81] *ERC-20 Token Standard*. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md

[82] *OpenZeppelin Contracts*. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol

[83] *CryptoKitties GeneScience*. [Online]. Available: https://etherscan.io/address/0xf97e0a5b616dffc913e72455fde9ea8bbe946a2b#code

[84] A. E. Gencer, S. Basu, I. Eyal, R. Van Renesse, and E. G. Sirer, "Decentralization in bitcoin and ethereum networks," in *International Conference on Financial Cryptography and Data Security*. Springer, 2018, pp. 439–457.

[85] S. Blackshear, Private Communication, 2020.

[86] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of cryptographic engineering*, vol. 2, no. 2, pp. 77–89, 2012.

[87] D. Johnson, A. Menezes, and S. Vanstone, "The elliptic curve digital signature algorithm (ecdsa)," *International journal of information security*, vol. 1, no. 1, pp. 36–63, 2001.

[88] *Go Ethereum*, accessed May 31, 2020. [Online]. Available: https://github.com/ethereum/go-ethereum

[89] P. Sheng, G. Wang, K. Nayak, S. Kannan, and P. Viswanath, "Bft protocol forensics," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, 2021, pp. 1722–1743.

[90] J. Li and D. Maziéres, "Beyond one-third faulty replicas in byzantine fault tolerant systems," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, 2007, pp. 10–10.

[91] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1041–1053.

[92] Z. Xiang, D. Malkhi, K. Nayak, and L. Ren, "Strengthened fault tolerance in byzantine fault tolerant replication," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 205–215.

[93] D. Kane, A. Fackler, A. Gągol, D. Straszak, and V. Zamfir, "Highway: Efficient consensus with flexible finality," *arXiv preprint arXiv:2101.02159*, 2021.

[94] A. Haeberlen, P. Kouznetsov, and P. Druschel, "Peerreview: Practical accountability for distributed systems," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 175–188, 2007.

[95] A. Haeberlen and P. Kuznetsov, "The fault detection problem," in *Proceedings of the 13th International Conference on Principles of Distributed Systems*. Springer, 2009, pp. 99–114.

[96] P. Civit, S. Gilbert, and V. Gramoli, "Polygraph: Accountable byzantine agreement." *IACR Cryptol. ePrint Arch.*, vol. 2019, p. 587, 2019.

[97] A. Ranchal-Pedrosa and V. Gramoli, "Blockchain is dead, long live blockchain! accountable state machine replication for longlasting blockchain," *arXiv preprint arXiv:2007.10541*, 2020.

[98] A. Stewart and E. Kokoris-Kogia, "Grandpa: a byzantine finality gadget," *arXiv preprint arXiv:2007.01560*, 2020.

[99] J. Neu, E. N. Tas, and D. Tse, "Ebb-and-flow protocols: A resolution of the availability-finality dilemma," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 446–465.

[100] Y. Aumann and Y. Lindell, "Security against covert adversaries: Efficient protocols for realistic adversaries," in *Theory of Cryptography Conference*. Springer, 2007, pp. 137–156.

[101] G. Asharov and C. Orlandi, "Calling out cheaters: Covert security with public verifiability," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2012, pp. 681–698.

[102] D. Boneh, M. Drijvers, and G. Neven, "Compact multi-signatures for smaller blockchains," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2018, pp. 435–464.

[103] H. V. Ramasamy and C. Cachin, "Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast," in *International Conference on Principles of Distributed Systems*. Springer, 2005, pp. 88–102.

[104] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography," *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[105] S. Micali, "Byzantine agreement, made trivial," 2018. [Online]. Available: https://people.csail.mit.edu/silvio

[106] J. Chen, S. Gorbunov, S. Micali, and G. Vlachos, "Algorand agreement: Super fast and partition resilient byzantine agreement." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 377, 2018.

[107] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*. IEEE, 1999, pp. 120–130.

[108] D. Association, "Diem," 2020. [Online]. Available: https://github.com/diem/diem

[109] S. Bano, A. Sonnino, A. Chursin, D. Perelman, and D. Malkhi, "Twins: White-glove approach for bft testing," *arXiv preprint arXiv:2004.10617*, 2020.

[110] M. J. Fischer, N. A. Lynch, and M. Merritt, "Easy impossibility proofs for distributed consensus problems," *Distributed Computing*, vol. 1, no. 1, pp. 26–39, 1986.

[111] A. Mostefaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous byzantine consensus with t< n/3 and o (n2) messages," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, 2014, pp. 2–9.

[112] D. Dolev and R. Reischuk, "Bounds on information exchange for byzantine agreement," *Journal of the ACM (JACM)*, vol. 32, no. 1, pp. 191–204, 1985.

[113] D. Dolev and H. R. Strong, "Authenticated algorithms for byzantine agreement," *SIAM Journal on Computing*, vol. 12, no. 4, pp. 656–666, 1983.

[114] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous byzantine agreement with expected $O(1)$ rounds, expected $O(n^2)$ communication, and optimal resilience," in *International Conference on Financial Cryptography and Data Security*. Springer, 2019, pp. 320–334.

[115] J. Katz and C.-Y. Koo, "On expected constant-round protocols for byzantine agreement," in *Annual International Cryptology Conference*. Springer, 2006, pp. 445–462.

[116] I. Abraham, K. Nayak, L. Ren, and Z. Xiang, "Good-case latency of byzantine broadcast: A complete categorization," in *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021, pp. 331–341.

[117] I. Abraham, T. H. Chan, D. Dolev, K. Nayak, R. Pass, L. Ren, and E. Shi, "Communication complexity of byzantine agreement, revisited," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 317–326.

[118] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[119] J. Neu, E. N. Tas, and D. Tse, "Snap-and-chat protocols: System aspects," *arXiv preprint arXiv:2010.10447*, 2020.

[120] J. Neu, E. N. Tas, and D. Tse, "The availability-accountability dilemma and its resolution via accountability gadgets," in *International Conference on Financial Cryptography and Data Security*, 2022.