NIST Internal Report
NIST IR 8460 ipd

State Machine Replication and Consensus with Byzantine Adversaries

Michael Davidson

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8460.ipd

NATIONAL INSTITUTE OF
STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Internal Report
NIST IR 8460 ipd

State Machine Replication and Consensus with Byzantine Adversaries

Michael Davidson
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.IR.8460.ipd

April 2023

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

Certain commercial equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at https://csrc.nist.gov/publications.

NIST Technical Series Policies

Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History

Approved by the NIST Editorial Review Board on YYYY-MM-DD [Will be added upon final publication]

How to cite this NIST Technical Series Publication:

Davidson M (2023) State Machine Replication and Consensus with Byzantine Adversaries. (National Institute of Standards and Technology, Gaithersburg, MD), NIST IR 8460 ipd. https://doi.org/10.6028/NIST.IR.8460.ipd

NIST Author ORCID iDs

Michael Davidson: 0000-0002-4862-5697

Public Comment Period

April 26, 2023 - September 1, 2023

Submit Comments

ir8460-comments@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Abstract

The objective of state machine replication (SMR) is to emulate a centralized service in a distributed, fault-tolerant fashion. To this end, a set of mutually distrusting processes must agree on the execution of client-submitted commands. Since the advent of Bitcoin, the idea of SMR has received significant attention. This document surveys both classical and more modern research on SMR and details many of the most significant permissioned and permissionless algorithms, their performance, and security considerations.

Keywords

atomic broadcast; Bitcoin; blockchain; Byzantine Fault Tolerance; consensus; cryptocurrency; distributed ledger technology; Ethereum; state machine replication.

Note to Reviewers

Figure 22a © IFCA 2014.
Figure 28 © IFCA 2015.
Figure 37 © IFCA 2020.
Figure 42 © IFCA 2018.
Figure 51 © IFCA 2021.
Figure 56 © IFCA 2020.

Table of Contents

List of Tables

List of Figures

Acknowledgments

The author thanks Tyler Diamond, Frederic de Vaulx, and Andrew Regenscheid for their helpful comments.

Executive Summary

Since the deployment of Bitcoin on January 3rd, 2009, and its description by the pseudonymous Satoshi Nakamoto in 2008 [1], research and development of new, practical state machine replication (SMR) systems have surged. It has been stated that Bitcoin provided a solution to the "Byzantine Generals Problem." While not strictly true, it is a useful starting point for this document's analysis of consensus algorithms, state machine replication, and distributed ledger technology (DLT).

More generally, the goal of these types of problems is to allow a set of mutually distrusting processes (e.g., computer processes) to agree on the outcome of some deliberation despite the possibility that some of them are faulty or even malicious. In essence, the goal is to provide some service to clients that emulates a centralized service while operating as a distributed server. There are a variety of ways to formulate this problem (several are described in Section 1), but they all require some notion of agreement between the distributed processes. Research in this area began in the early 1980s when the Byzantine Agreement [2] and Byzantine Generals [3] problems were formulated. The first algorithms to solve these problems were extremely inefficient, and real-world usage did not become plausible until the celebrated Practical Byzantine Fault Tolerance (PBFT) algorithm was invented in 1999 [4].

Until the advent of Bitcoin, it was believed that all distributed agreement algorithms required a fixed set of identifiable participants known in advance. Bitcoin was the first protocol to demonstrate that consensus can be maintained across distributed processes in an open network with free entry and no fixed identifiers. Today, the terms *permissioned* and *permissionless* are used to describe the difference between the two models.

Bitcoin also popularized the idea of a *blockchain* as the data structure over which the distributed processes maintain agreement. Since then, the concept has been generalized to *distributed ledgers* more broadly. A blockchain is an ordered list of client-submitted commands, or transactions, that modify the system state. Because all participants execute the same agreed-upon commands in the same order, participants are able to maintain a common view of the execution of a protocol-defined state machine. In addition to cryptocurrencies, state machine replication and the *smart contracts* they enable have been suggested for use in trade settlement, finance, identity management, supply chains, healthcare, Internet of Things (IoT), and other industries.

0.1. Purpose and Scope

This document is intended to serve as an advanced treatment of consensus algorithms, state machine replication, and distributed ledger technology. It may also function as a reference for consensus algorithms as it contains fairly detailed descriptions of a variety of algorithms that may be useful in different scenarios. The reader is expected to already have a high-level understanding of distributed ledger technology, such as that provided by NIST IR

8202, *Blockchain Technology Overview* [5].

This document first discusses the properties required of distributed consensus systems and the many kinds of subprotocols used to implement them in a variety of system models. Many algorithms, both permissioned and permissionless, are then described in detail. The discussion on permissioned consensus starts with the classic PBFT algorithm but focuses heavily on techniques that have been developed more recently and improve performance or enable security in more challenging environments. Permissionless algorithms are divided into categories based on the Sybil-resistance mechanism employed – that is, proof of work (PoW), proof of stake (PoS), and alternative mechanisms. There is extensive discussion on the unique security issues that arise in each case, the architectural reasons they exist, and techniques that can be used to mitigate them. Finally, a variety of more advanced topics are discussed, including scalability methods such as sharding and "layer 2" technologies, interoperability, state machine design, networking, and how incentives impact system security.

0.2. Notes on Terminology

The distributed systems literature is rife with synonyms and inconsistent or imprecise use of terms. When the word "consensus" appears in this document, it is meant as a generalization that captures all of the agreement problems described, including various broadcast problems, Byzantine Agreement, and state machine replication. The term "broadcast" – when not being used to describe broadcast problems specifically – is meant to convey the idea of simultaneously transmitting a message to multiple peers. In addition, the ill-defined term "decentralized," which is used frequently in the literature, is discussed in more detail in section 3.1.

Several groups of synonymous words appear in this document. Most of the time, terminology from the original source paper was used. For example, the terms "node," "replica," "process," "validator," and "miner" are used as synonyms but often in slightly different contexts, such as an entity that participates in consensus. Many protocols have leaders, which may be called the "primary" or "block producer." When a node is not the leader, it may be a "secondary" or "backup." A "malicious" node may be considered "Byzantine," "faulty," "corrupt," or "dishonest," whereas honest nodes are sometimes called "correct." When nodes eventually agree on a value, it is sometimes said that they "decide," "output," or "accept" the value.

The term "blockchain" is defined in NIST IR 8202, *Blockchain Technology Overview*:

> Blockchains are distributed digital ledgers of cryptographically signed transactions that are grouped into blocks. Each block is cryptographically linked to the previous one (making it tamper evident) after validation and undergoing a consensus decision. As new blocks are added, older blocks become more difficult to modify (creating tamper resistance). New blocks are replicated across copies of the ledger within the network, and any conflicts are resolved auto-

matically using established rules. [5]

While this is a good description of some specific blockchains, the term is used here to only mean "a chain of blocks," capturing the fact that blocks are cryptographically linked together in a list. This makes blockchains a particular data structure within a broader set of distributed ledgers.

0.3. Document Structure

The rest of this document is organized as follows:

- **Sections 1-3** are introductory material. **Section 1** introduces the formal problems that are solved by the protocols described throughout the document and some of the model assumptions under which the problems can be solved. **Section 2** describes the various sub-protocols and components that are often used in designing distributed ledger systems for SMR. **Section 3** discusses the trade-offs between maintaining "decentralization" and the scalability of DLT systems.

- **Sections 4-8** describe protocols for permissioned consensus. **Section 4** describes Practical Byzantine Fault Tolerance (PBFT), the first system design scalable enough to be used in practice. **Section 5** describes more modern, high-performance consensus algorithms. **Section 6** discusses algorithms that are designed for asynchronous networks where messages may be arbitrarily delayed. **Section 7** surveys a variety of extra properties that one might desire from a permissioned consensus algorithm. **Section 8** describes protocols where participants may select their own quorums of trusted replicas and need not be aware of the existence of every replica in the network.

- **Sections 9-11** discuss protocols that use proof of work (PoW) as a Sybil-resistance mechanism. **Section 9** discusses aspects common to most PoW protocols. **Section 10** describes Nakamoto Consensus – the protocol used by Bitcoin – in detail. **Section 11** describes a wide variety of PoW consensus designs.

- **Sections 12-13** discuss protocols that use proof of stake (PoS) as a Sybil-resistance mechanism. **Section 12** provides a historical overview of PoS and the security issues that need to be considered as part of PoS algorithm design. **Section 13** describes a variety of specific PoS protocols.

- **Section 14** discusses protocols that use alternative Sybil-resistance algorithms, such as proof of space, as well as hybrid mechanisms.

- **Sections 15-19** cover a variety of more advanced topics related to the design of DLT for SMR. **Section 15** discusses the technical details of one of the more promising scalability methods: sharding. **Section 16** discusses interoperability between systems. **Section 17** covers topics related to the network layer of these protocols, such as how a node discovers new peers and communication strategies. **Section 18** discusses state machine design considerations and some "layer 2" scaling protocols that can be built on an underlying replicated state machine. **Section 19** considers a variety of incentive-related security issues that can arise in these systems.

1. Introducing the Problems

1.1. The Byzantine Generals Problem

The *Byzantine Generals Problem* (BGP) is equivalent to the *Byzantine Broadcast* (BB) problem, which is the more commonly used term in the distributed systems literature. This problem was introduced in [3] as *interactive consistency*.

Consider a city besieged by several divisions of the Byzantine army. Each division is led by its own general, and the generals communicate with each other via messenger. The generals must agree upon a common strategy: either attack or retreat. Unfortunately for the Byzantine army, some of the generals may be disloyal and actively working to sabotage the agreement. A solution to the BGP, then, is an algorithm that ensures that:

- Every loyal general decides upon the same strategy.
- If the number of traitors is small, the traitors cannot cause the loyal generals to decide on a "bad" plan – that is, a plan they would not have otherwise agreed to in the first place.

Stated differently, let there be n total generals (or computer processes), up to f of which may be disloyal (or malicious/faulty). One general in particular, the commanding general, sends an order to $n-1$ lieutenant generals, such that the following interactive consistency conditions are maintained:

- **IC1:** All loyal lieutenants obey the same order.
- **IC2:** If the commanding general is loyal, then all loyal lieutenants obey the commander's order.

The same paper proved that the BGP is solvable only if more than $\frac{2}{3}$ of the generals are loyal when using only "oral" messages (unsigned messages) and solvable for any number of generals/traitors with "unforgeable written messages" (signed messages) when there is a known fixed upper bound on how long it takes to send a message from one general to another (synchrony). That is, in a synchronous network without using digital signatures, if f generals are traitors, then no algorithm will work without $n > 3f$ total generals. In a synchronous network with signed messages, the problem is solvable with $n > f$ generals.

1.2. Broadcast Problems and Byzantine Agreement

A variety of broadcast problems exist, including the BGP. In each case, there is a designated sender with an input value that they would like to distribute to the remainder of the processes. An algorithm that solves a broadcast problem will have certain properties, such as the "interactive consistency" requirements described in Section 1.1. In particular, there are requirements around *consistency* (also known as *agreement*), *validity*, *integrity*, and *termination*. The consistency, validity, and integrity properties are the same for most broadcast problems, but the termination property differs. The following are the properties of a Byzantine Broadcast (BB) algorithm, which is equivalent to BGP and has the strictest termination requirement:

- **Consistency/Agreement**: If two honest replicas decide v and v', then $v = v'$.
- **Validity**: If the sender is honest and begins with input value v, then all honest nodes decide v.
- **Integrity**: Every honest process delivers at most one value, and if it delivers v, then some process must have broadcast v.
- **Termination**: Every honest process eventually decides some value.

Notice that property IC1 from the previous section is equivalent to consistency, while IC2 is equivalent to the validity property. By relaxing the termination requirement, different broadcast problems can be described as follows:

- *Reliable broadcast* (RB or RBC) requires that either all honest processes eventually decide, or no honest process decides. That is, RBC may not terminate if the sender is faulty, but if any honest party obtains an output, all other honest parties must as well.
- *Byzantine consistent broadcast* (BCB) allows some honest parties to decide without requiring all honest parties to do so.
- *Terminating reliable broadcast* (TRB) requires correct processes to agree on the receipt of a message or agree that the sender is faulty (so termination must occur, but correct processes need not get a value out of it).

Reliable broadcast is commonly used as an underlying communication primitive for more complex distributed protocols, such as multi-party computation (MPC) or consensus. Messages "delivered" by the broadcast algorithm are then used as input messages in an MPC or consensus protocol execution. Note that reliable broadcast does not guarantee agreement over the order of messages, only that the messages are delivered at all.

While broadcast algorithms are only mentioned a few times in this document, they have many similarities to the algorithms discussed throughout while being fairly easy to understand. Bracha's broadcast is among the most celebrated of reliable broadcast algorithms and was originally proposed in the late 1980s as a technique to convert crash fault tolerant consensus protocols into ones capable of resisting malicious adversaries, often called *Byzantine* adversaries [6].

1. The designated sender with starting value v sends a message, $(INIT, v)$, to all processes (here, $n = 3f + 1$). Then each process executes the following three steps.
2. Wait until the receipt of either one $(INIT, v)$ message, $2f + 1$ $(ECHO, v)$ messages, or $f + 1$ $(READY, v)$ messages for some v. Then send $(ECHO, v)$ to all processes.
3. Wait until the receipt of $2f + 1$ $(ECHO, v)$ messages or $f + 1$ $(READY, v)$ messages for some v, including messages received in the previous step. Then send $(READY, v)$ to all processes.
4. Wait until the receipt of $2f + 1$ $(READY, v)$ messages for some v, including ones received in the prior steps. Then deliver v.

Bracha's broadcast algorithm is a solution to the reliable broadcast problem so long as $n \geq 3f + 1$ and protocol messages eventually reach their destination. Consider the following arguments:

1. Let p and q be two correct processes, and assume that process p is the first to send a $(READY, v)$ message, while process q is the first to send a $(READY, v')$ message. Assume (by contradiction) that $v \neq v'$. Since process p is the first to have sent $(READY, v)$, it must have seen at least $2f + 1$ $(ECHO, v)$ messages. Similarly, process q must have seen at least $2f + 1$ $(ECHO, v')$ messages. In total, $4f + 2$ $ECHO$ messages were sent. Because $n \geq 3f + 1$, at least $f + 1$ replicas must have sent both $(ECHO, v)$ and $(ECHO, v')$ messages. However, this implies that an honest replica sent both an $(ECHO, v)$ and an $(ECHO, v')$ message. Because honest replicas do not equivocate (they only send one message of each type), this is a contradiction. Therefore, $v = v'$.
2. Now assume that process p delivers a value v, and process q delivers a value v'. This means that process p saw at least $2f + 1$ $(READY, v)$ messages, of which at least $f + 1$ are from honest replicas. Analogously, process q must have seen at least $f + 1$ $(READY, v')$ messages from honest replicas. By argument 1, this implies that $v = v'$.
3. If an honest process p delivers the value v, then every other honest process will eventually deliver v. Because p accepts v, it must be the case that p has seen $2f + 1$ $(READY, v)$ messages, and at least $f + 1$ of those were from honest processes. This implies that every other process will see at least $f + 1$ $(READY, v)$ messages and then send their own $(READY, v)$ message (because honest processes cannot equivocate). Therefore, at least $n - f \geq 2f + 1$ processes send a $(READY, v)$ message, so all honest processes will eventually see $2f + 1$ $(READY, v)$ messages and deliver v.
4. If the designated sender p is honest and broadcasts v, all honest processes eventually

> decide v. Every honest process will receive the sender's $(INIT, v)$ message and then send an $(ECHO, v)$ message. Then every honest process q will receive $n - f \geq 2f + 1$ $(ECHO, v)$ messages from other honest processes (and at most f other messages from faulty processes). Process q will then send a $(READY, v)$ message. In the final step of the protocol, all honest processes will receive $n - f \geq 2f + 1$ $(READY, v)$ messages (and at most f ready messages from faulty processes). Therefore, honest process q will deliver v.

Putting this all together, argument 4 shows that if the designated sender p is honest and broadcasts v, all honest replicas will deliver v. On the other hand, if the designated sender p is faulty and some honest process q delivers v, then all honest processes deliver v by argument 3. Otherwise (p is faulty and q does not accept v), no honest process will accept any value. This proves that Bracha's algorithm achieves reliable broadcast.

Byzantine agreement (BA), introduced in [2], is a closely related problem to broadcast but with one crucial difference: instead of having a dedicated sender who disseminates a value to the rest of the network, every process starts with an initial value. This redefined problem has many synonyms in the literature, including *consensus*, *total order broadcast*, and *atomic broadcast*. Often, the term "consensus" is used specifically for one instance of BA (*single-shot BA*), while "atomic broadcast" is used in repeated or sequential BA, where the agreed upon values are also in an agreed upon order. This document uses the terms interchangeably but primarily focuses on repeated BA, which is needed for state machine replication. When BA is used to agree on a single bit, it is called *binary Byzantine agreement*, whereas it is called *multi-value Byzantine agreement* (MVBA) when there are more than two possible choices.

The consistency, integrity, and termination properties are the same in BA and BGP, but the validity property necessarily changes to reflect the lack of a designated sender. Specifically, the validity property of BA is that if all correct processes propose the same value v, then any correct process must decide v. There can also be *weak validity* (in contrast to the "strong" validity just mentioned): for each correct process, its output must be the input of some correct process. In either case, validity may also require the decided value to satisfy an external predicate, like a valid digital signature or other (state machine) rules. In this document, most protocols aim for strong validity.

1.3. State Machine Replication (SMR)

State machine replication is built off of atomic broadcast, which guarantees that each agreed-upon value is also in an agreed-upon order. These values are typically called *transactions* or commands. Processes send these transactions to each other, which are then placed in a total ordering via atomic broadcast and then executed by the processes in that order. Transactions operate on some global state and transform the state via a deterministic program. SMR guarantees lock-step execution of identical commands and agreement over state by all honest processes. The SMR approach was first described in [7] but was popularized in [8].

Distributed ledger technology (DLT) and blockchains are specific examples of state machine replication. Given a blockchain protocol, one can derive the SMR by having replicas execute the blockchain protocol, having honest nodes broadcast all transactions they see to each other, and – for Bitcoin and many related protocols – removing some number of trailing *blocks* (sets of transactions). Such a distributed ledger provides the following properties [9]:

- **Persistence**: Once a transaction is at least k blocks deep into the ledger of an honest node (where k is a security parameter), it will be included in the same permanent position in the ledger of every honest node with overwhelming probability.
- **Liveness**: All transactions from honest clients will eventually be at a depth of more than k blocks in an honest node's blockchain.

A distributed ledger can be shown to satisfy persistence and liveness if it satisfies the *common prefix*, *chain quality*, and *chain growth* properties, which are discussed in more detail in Section 10.1. Informally, the common prefix property says that the blockchains of two honest nodes differ only in their last k blocks from the chain tip. Chain quality is the property that "enough" of the blocks that wind up in the blockchain were proposed by honest nodes. Finally, chain growth means that the blockchains accepted by honest nodes continuously grow at a certain pace.

It may be tempting to think that state machine replication can be realized by simply executing a BA protocol repeatedly in serial, but this is false due to an issue with how BA's validity property is defined [10, 11]. In BA, validity ensures that if all honest replicas had a particular input value, then that is the value of the output as well. This does not suffice for SMR, where each replica maintains a local buffer for transactions they have received but have not agreed upon yet (often called the *mempool*), and it cannot be guaranteed that honest parties will initiate BA with the same transaction set. As a result, achieving validity would likely come at the expense of liveness, as it would be challenging for honest replicas to agree on anything other than empty blocks. This is not a problem for distributed ledgers, which can base their validity on enforcing the predicate that whatever state machine rules exist must be followed for a block to be considered valid.

One of the primary use cases for SMR is payments, or secure asset transfer. For this specific use case, enforcing a total order on transactions is not strictly necessary; some form of reliable broadcast is sufficient [12]. As long as user accounts are associated with distinct owners, then the owner of an account determines the order of transfers out of their account with no need to agree on the ordering with other users. Others only need to verify that the owner's decisions maintain any needed causal relations among accounts (e.g., that the order does not create a scenario where an account transfers out more units of an asset than it possesses at the time). These causal relations establish a *partial ordering* rather than a total ordering. Several payment systems based on this idea have been proposed, and they are usually highly performant by eliminating the requirement of agreeing on a fully linearized ordering of transactions [13–19].

Relying on a partial ordering has trade-offs. Most notably, without enforcing a total ordering on transactions, the majority of complex smart contracts (e.g., on-chain cryptocurrency exchanges) are no longer possible. If transactions are partially ordered, they may attempt to access and modify the same system state concurrently, which would lead to inconsistencies. Another consequence of replacing BA with reliable broadcast is that the termination property of reliable broadcast is not guaranteed if the designated sender is faulty. In this case, it means that if the spender in a transaction submits two conflicting payments to the network, it is possible that neither transaction is ever executed. This is not a problem if one assumes that a client who attempts to double-spend in this way is malicious. However, in practice, this behavior may be non-malicious, in which case an innocent user may have their funds frozen permanently as honest nodes fail to agree on a transaction to include in the ledger. For example, when the Bitcoin network is congested, some clients will use a technique called *replace-by-fee* (RBF) to reissue a transaction with a higher fee attached with the intention of "jumping the line" and having miners include the transaction in the blockchain more quickly. A payment system with partial ordering is unable to handle this scenario.

1.4. The Adversary

When discussing the security of various consensus protocols, it is important to consider the powers that an adversary has available to disrupt the protocol. It is generally assumed that the adversary in a distributed system controls f parties and has the ability to coordinate them. For instance, in a protocol that has $n = 10$ parties and can tolerate up to $f = 3$ faulty ones, a single adversary controls all faulty parties (or faulty parties are in collusion). This assumption is favorable to the attacker and conservative for the protocol because – in the real world – the faulty parties could be controlled by multiple adversaries. That would correspond to a situation where, say, each of the three faulty parties were hacked by a different adversary. If the protocol is secure against the combined adversary, it would also be secure against a less coordinated set of adversaries.

The adversary can be either *static* or *adaptive*. In the static model, the adversary corrupts its f parties at the beginning of the protocol execution, and those specific parties remain corrupt for the duration of the execution. Alternatively, an adaptive adversary can observe the protocol execution and corrupt parties "on the fly." That is, one can think of the adversary as having a "corruption budget" of f, which can be used throughout the protocol execution and chosen to the adversary's advantage based on the messages seen. Attaining provable security against an adaptive adversary is significantly more challenging than against a static adversary.

Various models exist regarding what specific powers the adversary has over the corrupted replicas. The two most common types of faults that a protocol may be designed to tolerate are *crash faults* and *Byzantine faults*. In a crash fault, the faulty party simply stops participating in the protocol. This corresponds to victims of denial-of-service attacks as well as more benign crashes. It is assumed that these parties never recover. In this document, the focus is on Byzantine faults and Byzantine fault-tolerant (BFT) protocols as opposed to crash fault-tolerant (CFT) ones. A Byzantine fault is any arbitrary fault, including deliberately malicious ones, such as sending equivocating messages to try to disrupt the protocol. Other possibilities exist that are outside of the scope of this document, such as *omission faults* (not sending messages when supposed to), and crash failures *with* recovery. Faults of each type can also be caused by software bugs rather than any particular malicious action.

Protocol designers often focus on two types of participants: 1) the honest ones, who faithfully execute the protocol as specified no matter what, and 2) the Byzantine ones, who are controlled by the adversary and deliberately attempt to arbitrarily subvert the protocol (whether this is beneficial to the adversary or not). However, there may be cases in which honest behavior is irrational and not in the best interest of the participants. The assumption that a particular fraction of the participants behave honestly may be unlikely to hold in practice. Traditional security models typically ignore this.

In contrast, the BAR model [20] considers three types of players: 1) Byzantine players; 2) altruistic players, who always behave honestly even if it is irrational for them to do so; and 3) rational players, who will act selfishly and deviate from the protocol in order to increase their utility but will not arbitrarily deviate. This is a more challenging model for a protocol to be secure against because it must also be in the best interest of players to behave honestly based on some utility function. This involves the use of game-theoretic techniques that are beyond the scope of this document. It is also possible to design protocols without any altruistic players [21].

1.5. Timing Assumptions

Among the most important factors in evaluating a consensus protocol are the assumptions it makes on the timing of message delivery. Naturally, more conservative timing assumptions make the protocol more robust but impose stricter requirements on its design and may negatively impact performance. In each case, it is assumed that if an honest party sends a message to another honest party, it is eventually received, even though it may take a while or arrive out of order (perhaps an adversarially chosen order). Here, time is usually measured in terms of rounds of protocol execution. The three most common network timing models are *synchronous*, *asynchronous*, and *partially synchronous*.

- **Synchronous**: There is a fixed known upper bound, Δ, on the time it takes for messages to be sent from one processor to another. If an honest replica sends a message to another honest replica in round r, then the recipient will have seen the message by, at latest, the beginning of round $r+\Delta$. In synchronous networks, there can be a *rushing adversary* who acts last in each round of the protocol and can see all messages sent by honest parties before deciding what to do with their control over corrupted participants. Under synchrony, agreement is possible when the majority of participants are honest ($n \geq 2f+1$), while broadcast simply requires that $f < n$, assuming a PKI and digital signatures.

- **Asynchronous**: There is no fixed bound on the time it takes for messages to be sent from one honest processor to another. There may or may not be guaranteed delivery eventually, but if not, consensus is impossible due to the important "FLP impossibility" result [22]. If messages are guaranteed to be delivered eventually, then secure protocols can be designed. The optimal resilience for asynchronous consensus protocols is $n \geq 3f+1$, and safety and liveness are both maintained so long as this holds. Progress occurs as messages arrive rather than based on fixed rounds.

  The celebrated FLP impossibility result shows that no asynchronous and deterministic protocol can achieve consensus with even one faulty processor, and this fault need not be Byzantine. A simple crash can prevent consensus due to the inability to ensure that the protocol will terminate. This result can be circumvented in two ways: by using randomization as part of the algorithm [23, 24] or by providing probabilistic termination (with probability 1) [25]. Probability 1 does not imply that all executions terminate, but that the set of non-terminating executions is extremely improbable in the limit. The difference between probabilistic termination and using randomization is that the protocol itself does not incorporate randomness in probabilistically terminating protocols.

  Asynchronous protocols may be even easier to implement than synchronous or partially synchronous protocols because they do not require dealing with explicit timeouts. The implementation can be entirely message-driven.

- **Partially synchronous**: This model was introduced in [26], which presented two conceptions of the idea:

  1. There is a fixed but unknown upper bound, Δ, on the time it takes for messages to be sent from one process to another. This is sometimes called *semi-synchronous* or the *bounded delay* model.
  2. There is a fixed, known upper bound, Δ, for message transmission, but this bound is only guaranteed to hold after some unknown time called the *Global Stabilization Time* (GST). That is, there is a period of asynchrony (where messages may be lost) followed by a period of synchrony.

  Since messages can be lost prior to GST, they must be re-sent every round to ensure that they are eventually received. This implies that there may be unbounded communication costs prior to GST [27]. In the bounded delay variant, messages cannot be lost, so only a single message transmission is needed.

  This model decouples the safety/consistency and liveness properties of the system. If the underlying network is in fact asynchronous, then a partially synchronous protocol will continue to maintain safety but lose liveness and stall. The optimal resilience for partially synchronous protocols is $n \geq 3f+1$, and safety is maintained so long as this holds.

Synchronous consensus algorithms are desirable for their superior security bounds compared to asynchronous and partially synchronous protocols ($n \geq 2f+1$ for synchronous but $n \geq 3f+1$ otherwise). However, synchronous consensus protocols are inconvenient for at least two other reasons. First, latency is dependent on the estimated network delay bound Δ, which creates a trade-off between security and latency (*responsive* protocols can solve this; see Section 5.2). Second, synchronous protocols are unable to tolerate network partitions, which is problematic for long-running protocols over the internet.

The *sleepy model* is meant to capture the beneficial security bounds of synchronous protocols while making the system more partition-tolerant [28]. In other models, honest nodes are assumed to be online throughout the entirety of the execution. Once a node goes offline, it is considered faulty forever, even after coming back online. In contrast, the sleepy model allows nodes to be "alert" (online) or "asleep" (offline), where asleep nodes can wake up and become honest again. Synchronous protocols have difficulties because if an honest player is offline for a sufficiently long time ($> \Delta$), they will reject honest messages when they come back online. Asynchronous protocols require a threshold of honest validators to respond to a proposal, but there may not be that many honest parties online at the time. The sleepy model is similar to synchrony in that alert nodes have a network with a known delay parameter and similar to asynchrony in that nodes are allowed to go offline indefinitely and receive all pending messages upon coming back online. The sleepy model allows consensus as long as a majority of the alert nodes are honest. That is, it has the same security bound as synchronous systems when everyone is online and scales down as replicas go offline. Even if only a tiny fraction of the system's nodes are alert, the protocol will continue to make progress; that is, the sleepy model aims to maintain liveness despite an arbitrary number of nodes going offline.

A related model, *weak synchrony*, has also been proposed as a way of dealing with nodes temporarily going offline [29]. In contrast to the sleepy model, weakly synchronous protocols favor consistency rather than liveness. When the network is partitioned, the minority partition stops making progress but avoids the risk of deciding on values or blocks that are inconsistent with the majority partition. The weak synchrony assumption is that the majority of nodes are both honest and online in each round, but the set of honest and online nodes need not be the same in each round. This is a generalization of the synchronous model, where the honest and online set must contain every honest node in every round. Most synchronous consensus protocols can adopt slight adjustments in order to be secure under weak synchrony [30].

Not every problem is solvable in any given model. For example, Byzantine broadcast only works under synchrony, not partial synchrony or asynchrony, even if at most one replica is faulty. The termination property cannot be satisfied because replicas do not know whether the sender sent them a message at all, and thus do not know whether to keep waiting for it or use a default value. The sender can be faulty and simply not send a message. To get around this, one must weaken the termination property, as is done for reliable broadcast.

1.6. Permissioned vs. Permissionless

After the advent of Bitcoin, it became important to draw a distinction between the classical consensus protocols designed since the 1980s (permissioned) and the new style of protocol (permissionless).

In the *permissioned* model of consensus, there are n replicas, up to f of which may be under the control of the adversary. Both n and the identity of the replicas are known to every participant. Communication between participants typically takes place over authenticated channels, in which case the existence of a public key infrastructure (PKI) is generally assumed. At a minimum, every replica needs to agree with every other replica about the set of public keys used in the system. The permissioned model corresponds to classical consensus algorithms, which are discussed in Sections 4 through 7.

Permissionless systems differ from classical ones in four key ways, according to Pass and Shi [31]:

1. There is no access control mechanism that determines which nodes can join the system, and nodes can freely join or leave the system at any time.
2. Nodes are not aware of the other protocol participants a priori. In particular, communication is not over authenticated channels, so message senders are not authenticated.
3. The protocol itself may be unaware of how many nodes are participating in its execution.
4. The number of nodes involved in the system can grow or shrink over time.

Pass and Shi go on to prove that several of the limitations of permissionless consensus are required in order to work in such a challenging environment [31]. First, a Sybil-resistance mechanism is needed in order to maintain consensus when communication is not authenticated (Sybil-resistance mechanisms are introduced in Section 2.2; the proof-of-work mechanism is assumed in [31]). This is because an adversary needs to have their messages rate-limited, regardless of whether nodes can join freely, everyone knows how many nodes there are, and without message delays. Second, in order to allow nodes to freely join the system after it has been set up, proofs of work must be performed continuously throughout the lifetime of the system. If proofs of work ever cease, then new nodes can be tricked into preferring a simulated execution. Not all Sybil-resistance mechanisms can fully accomplish this free entry property (in particular, proof-of-stake systems require workarounds for this, as described in Section 12.1.2). To maintain this free entry condition, there must be an honest majority in control of the Sybil-resistance resource. This is a standard consistency argument, which is discussed more in Section 10.1. Finally, to tolerate uncertainty in the number of participants, there must be a known upper bound in the network delay. That is, a synchrony assumption is needed. This is true because the network can be partitioned in half, and if the adversary can cause an arbitrarily large message delay, there is always some delay that will cause a consistency violation. Honest nodes are unable to tell whether the other side of the partition exists at all or if there is just a message delay. This is discussed more thoroughly in Sections 10.1 and 10.2.1.

The permissionless model also has setup assumptions. In particular, permissionless networks require a trusted setup to create the *genesis block* – a data structure that encodes the initial state of the system. This is to prevent precomputation attacks, where the entity that creates the system creates a hidden blockchain in advance which can be used to gain an advantage in consensus. Famously, Satoshi Nakamoto included a newspaper headline in the Bitcoin genesis block in order to prove that this attack had not taken place.

2. System Components

Distributed ledger systems are usually composed of a variety of different subprotocols or components. Not every system will use each of the components described here. Further, a single subprotocol may be responsible for multiple aspects of the system simultaneously. Some of these subprotocols include:

- Data structures over which consensus is maintained
- Sybil-resistance mechanisms
- Leader election and/or committee selection
- Fork-choice or chain-selection rules
- Networking components
- Incentive mechanisms
- Cryptographic primitives
- The state machine itself

This section will introduce these ideas, but most will be explained in much more detail later in the document. Note that many systems in the literature describe only one or two of the components listed above but can often be adapted to alternative situations. For example, some fork-choice rules will be analyzed for a single Sybil-resistance mechanism but could be paired with a different mechanism in practice.

2.1. Data Structures for Distributed Ledgers

There are a variety of ways to organize the linearized/ordered transaction log in SMR. The simplest possibility is an ever-expanding list of transactions. It is typically more efficient to batch transactions into "blocks" (a group of transactions) rather than to handle them individually, which increases throughput at the expense of latency. These blocks can be "chained" together via collision-resistant cryptographic hash functions in order to form a blockchain. In other words, a blockchain is a chain of blocks that each reference a hash of the earlier blocks. In this way, each block is like a vote or commitment to the entire chain before it. In computer science terms, this is a singly linked list. Generally, there is a *block header* that includes metadata like the hash reference to the previous block and a commitment to the content of the block itself.

Some systems may incorporate multiple blockchains that operate in parallel and may provide different functionalities. Some systems employ a sharded architecture, where multiple blockchains exist in parallel but are coordinated via another blockchain. See Section 15 for more information on sharding. Other systems use directed acyclic graphs, or DAGs [32]. Technically, a blockchain is a simple DAG. However, the term is typically used in this space to describe systems where each block may point to multiple earlier blocks instead of just one. In a number of systems, there is a primary pointer to a previous block plus several so-called *uncle blocks*, which are produced around the same time but not in the "main" blockchain. Sometimes, the DAG is formed over individual transactions rather than blocks, which may result in only a partial ordering of transactions instead of a total ordering. There may also be a subprotocol for extracting a total order of transactions from the DAG.

The motivation for using a DAG or parallel blockchains is generally to improve latency and throughput (under optimistic assumptions) compared to the original blockchain structure. This is because proof-of-work blockchains require block distribution times to be a small fraction of the maximum network delay (see Section 10.2.1). Proof-of-work protocols that use DAGs are discussed in Section 11.5, and proof-of-stake DAGs are discussed in Section 13.3.

Note that throughout this document, the terms "blockchain" and "chain" are sometimes used generically to refer to any ledger. An example of a blockchain and a DAG are shown in Figure 1.

2.2. Sybil-Resistance Mechanism

A Sybil attack is a scenario in which a single real-world entity controls multiple in-protocol participants while making it look as though they are controlled by several different entities. Designers of permissioned networks need not be concerned with handling Sybil attacks because all replicas are already aware of each others' identities. However, in permissionless networks, a single malicious entity can represent itself as multiple distinct identities within

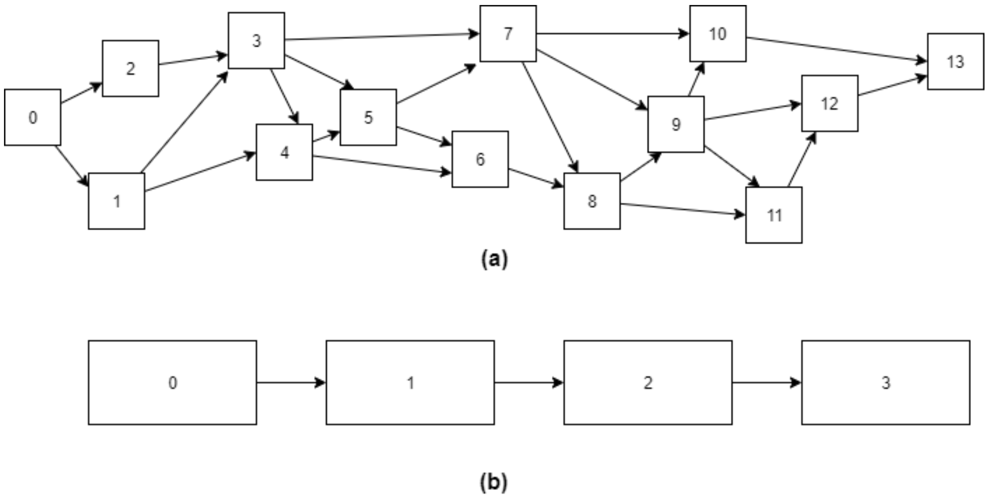


Fig. 1. Blockchain vs. DAG. Time moves from the left to the right. (a) A DAG where each block references two previous blocks except for the genesis block and the ones immediately after. In some systems, one of the pointers is the primary reference, while the other pointer is to an uncle block. (b) A simple blockchain.

a system. Therefore, permissionless systems require some method of addressing Sybil attacks.

Sybil attacks were introduced in [33]. Without a trusted central identification authority, Sybil attacks are always possible absent unrealistic or challenging assumptions, like resource parity among entities. However, permissionless distributed ledger systems can use some type of scarce resource to mitigate Sybil attacks. The two most commonly proposed Sybil-resistance mechanisms are proof of work and proof of stake. The scarce resource used under proof of work is computational effort, while the resource in proof of stake is virtual currency units that are native to the system. Proof of work and proof of stake have very different properties, which are discussed in detail in Sections 9 and 12. These differences can have material impacts on the security of a system, so designers cannot simply treat them as substitutes. There are other less common mechanisms as well, including proof of space (see Section 14.1) and those that utilize trusted execution environments (TEEs) [34, 35].

2.3. Leader Election and Committee Selection

Most consensus protocols require periodically electing a leader or a committee to perform particular tasks, such as proposing blocks of transactions to the rest of the network or voting on whether to accept a block that has been proposed. Other algorithms are leaderless, though this is less common.

In permissioned networks, the leader election subprotocol can be as simple as rotating through each validator in round-robin fashion. A typical method would be to label each of the n validators with a number in $\{0,...,n-1\}$, and for each round r, the leader is validator $r \mod n$. Alternatively, the leader could be chosen pseudorandomly based on a keyed hash function that hashes to the domain $\{0,...,n-1\}$. In some cases, the elected leader may remain the leader for many rounds, which can improve the network's throughput but at the cost of poor load balancing and fairness.

Leader election in permissionless networks is typically more complicated due to the lack of stable, known identities. In proof-of-work systems like Bitcoin (and its consensus algorithm, dubbed Nakamoto Consensus), the elected leader in a given round is the first node to solve a moderately hard puzzle. Proof of stake often uses advanced cryptographic primitives like verifiable random functions to aid in leader election, as described in Section 2.7. The process relies on uniformly random sampling from the participants in the network based on their share of the resource used for Sybil resistance. That is, a proof-of-work miner with 20% of the total computational power deployed on the network should have a 20% chance of being elected for each block. Leader election is typically tied to the minting mechanism for cryptocurrencies: the elected leader collects the newly created rewards (however, they can be decoupled [36]). As a result, leader election tends to have important implications for the incentive compatibility of a system.

If the leader is Byzantine, the outcome depends heavily on the specific protocol in use. In permissioned systems, as well as a few permissionless ones, a *view change* subprotocol is needed to securely move to the next leader (a *view* is a phase of the protocol where a particular replica acts as the leader; see Section 7.6). This is critical for maintaining liveness because a faulty leader can stall progress by not proposing a block when it is their turn. Progress in SMR is guaranteed once all correct processes synchronize to the same round and the leader of that round is correct. In most permissionless systems, a malicious leader can also weaken liveness by not including transactions in blocks and can sometimes use their status as leader to try to launch other attacks.

2.4. Fork-Choice or Chain Selection Rules

If a node is confronted with multiple valid ledgers, it has several methods for deciding which one to adopt, such as following the blockchain that has the most total proof of work (Nakamoto Consensus, see Section 10), selecting the heaviest subtree (GHOST, see Section 11.2), or accepting the result of a Byzantine agreement algorithm run by a duly elected committee of validators.

Nakamoto Consensus uses a simple fork-choice rule called the *longest chain rule*. When presented with two valid blockchains, a node in a system that uses Nakamoto Consensus will prefer the chain that has the largest cumulative amount of proof of work supporting it. For instance, if presented with the two chains from the bottom panel of Figure 2, a node will adopt the chain that ends with Block 5, assuming that each block has the same amount of work. Any alternative blocks that are discarded are called *stale blocks* or *orphan blocks*. If the node had adopted the other chain beforehand, the switch is called a *blockchain* *reorganization*, or *reorg* for short. In the top panel of Figure 2, the disconnected Block 2 and Block 4 are stale, and the shorter chain in the bottom panel that ends with Block 4 is stale. Note that transaction recipients can choose to wait for as many blocks as they want (called *confirmations*) before accepting a payment and providing goods or services, and waiting longer can offer more confidence that a block containing the payment will not become stale.

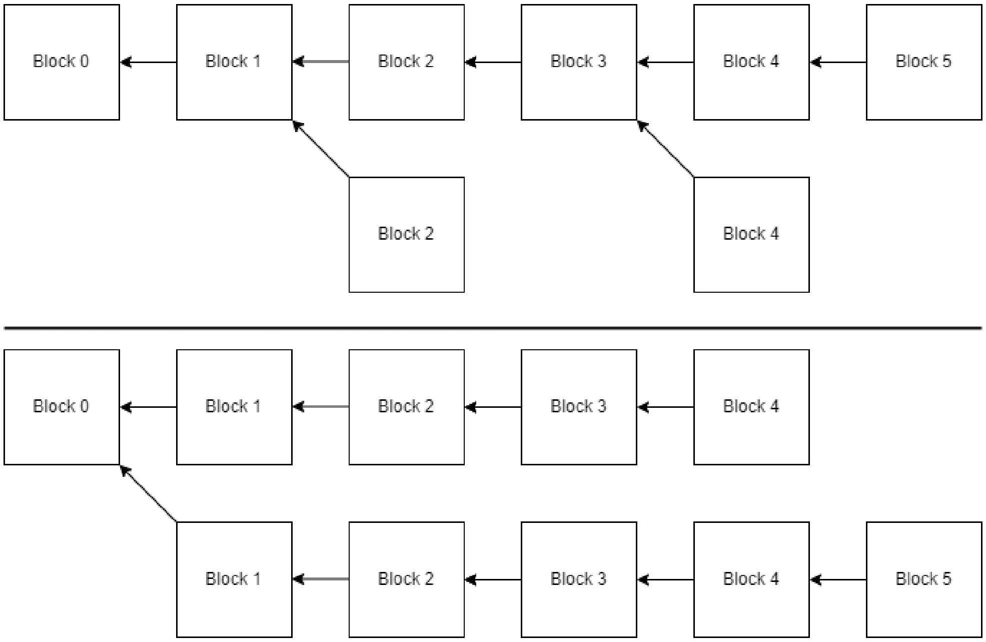


Fig. 2. Forking and the Longest Chain Rule.

2.5. Networking

In permissioned networks, validators will typically have pairwise point-to-point channels with all other validators, though not all systems are fully connected in this way. For example, in some systems (e.g., PBFT), every honest replica sends messages to every other replica for each round of the protocol. In other systems, replicas send messages to a leader, who then aggregates them and disseminates them to the rest of the replicas. Some systems even use a more complicated tree-like network topology for communication [37].

In permissionless settings, parties usually "diffuse" messages via gossip. Messages received by a party are then forwarded on to their peers with no specific source or destination for the message and without using authenticated channels. There is, thus, an implicit echoing assumption that messages are forwarded when heard. For permissionless networks, a more thorough treatment of possible networking components is provided in Section 17.1.

2.6. Incentive Mechanism

Historically, incentives were rarely considered in deployments of SMR. The replicas were typically run by a single organization that merely wanted additional fault tolerance. More recently, however, these systems have been deployed by a consortium of organizations or over open networks. In these cases, it is important that validators are incentivized to behave honestly rather than to try to disrupt the network.

In permissioned networks, honest behavior may be enforceable by appealing to legal systems. Permissionless systems do not have this luxury, so incentives must be considered an explicit part of the design. These incentives frequently involve some *token* native to the network (e.g., bitcoin in the Bitcoin network or ether in the Ethereum network). Typically, appending a new block to the blockchain gives the leader a reward that includes newly created cryptocurrency (the *block subsidy*) and transaction fees. In Bitcoin and many other systems, this block reward comes in a special transaction called the *coinbase* transaction, which is located first in a block and is the only transaction that can create new tokens rather than just transfer existing tokens. Due to the inconsistencies that can arise due to forking, the coinbase transaction has a *maturity window* in which funds cannot be spent from it (in Bitcoin, the maturity window is 100 blocks, or slightly under 17 hours).

Incentives are discussed in more detail in Section 19.

2.7. Cryptographic Primitives

This document assumes that the reader is familiar with basic cryptographic primitives like digital signatures and collision-resistant hash functions, which are used in all of the protocols described here. Beyond this, more advanced cryptography is applied judiciously in systems that implement state machine replication. This can be in either the atomic broadcast/consensus protocol or the state machine itself. Cryptographic techniques used in blockchain systems are surveyed in [38].

Hash functions are used in blockchain systems in a number of areas, such as for proof of work, generating *addresses* from public keys (concise representations of a public key or payment destination), and as building blocks for more advanced primitives, such as Merkle trees. Digital signatures are used to authorize transactions, implement authenticated channels, and vote in a variety of permissioned and proof-of-stake systems.

Threshold cryptography is frequently employed to improve fault tolerance or performance. Threshold signatures are the most common and are used both as a form of multi-factor authentication for signing transactions and to reduce the communication complexity of consensus by aggregating the individual signatures of several validators. Some protocols use threshold decryption to hide the contents of transactions from validators and prevent censorship or favoritism before transactions are ordered, only allowing them to be decrypted after being committed to the transaction log.

Cryptographic *accumulators* appear in nearly every blockchain system. Accumulators are compact ways of representing sets, which – at a minimum – allow concise proofs of membership or inclusion in the set. Block headers typically include an accumulator that commits to the set of transactions included in the block itself. The most common example of this is a *Merkle tree*, where each leaf of the tree is a transaction included in the block, and the root of the tree is included in the block header. Among other things, this allows *light clients* (see Section 3.2), which do not fully verify the ledger, to verify that transactions they care about have been added to the blockchain. In some systems, accumulators are also used to represent the global state of the system's state machine. That is, the block header may include an accumulator that includes all accounts and their balances.

Cryptographic *commitment* protocols are another common occurrence, though they are more commonly used in the state machine rather than in consensus. Commitments are a digital version of a sealed envelope – the party that commits to a value cannot alter the contents after the fact (it is already in the envelope), and no other party can discern the value until the committing party reveals it (no one can read the contents of the envelope until it has been opened). Cryptographic commitments are sometimes used to provide confidentiality for the amount sent in a transaction. This is often done by using commitments as a component of a *range proof*, which shows that a value is in a particular range and which can prove that a transaction did not create assets "out of thin air" to spend.

Zero-knowledge proofs – and particularly, *zkSNARKs* – are used in multiple ways in blockchain systems [39]. A zero-knowledge proof is a protocol that enables a prover to convince a verifier that a particular statement is true without the verifier learning anything other than the truth of the statement. zkSNARKs allow a prover to convince a verifier that the results of a computation on secret inputs are correct without the verifier needing to execute the computation or learn anything about it. This can be used to add privacy for payments but is also used in several advanced state machine designs, as discussed in Section 18.1.2. While zkSNARKs are powerful, they tend to require a trusted setup to generate parameters for the system, and an entity that knows the random values used for the setup has the power to create fraudulent proofs.

For the purposes of this document, the most important cryptographic primitives are those that aid in distributed randomness generation, which is most importantly used for leader election where an agreed-upon source of randomness must be shared among replicas. These schemes include common coin protocols, verifiable random functions (VRFs), and verifiable delay functions (VDFs).

A *common coin* is a randomness source that is observable by all participating processes but unpredictable for an adversary. The common coin abstraction can be realized using threshold signature schemes or *verifiable secret sharing*. Secret sharing allows a party to distribute a secret value to other parties such that a threshold of them are required in order to reconstruct the secret, while verifiable secret sharing uses commitments to ensure that the party who distributed the values has done so correctly. Sometimes, a *weak common coin* is sufficient, where weak means that there is a constant probability that the functionality will return different values to different processes.

Verifiable random functions (VRF), introduced in [40], are pseudorandom functions that provide publicly verifiable proofs of correctness. They can be thought of as a public key version of a keyed hash function. For a fixed key pair and input value, a VRF will produce a unique pseudorandom and verifiable output, even if the key pair was chosen adversarially. The VRF proof is usually the signature over some input data, and the pseudorandom output is a hash of the signature. A verifier will ensure that the signature is valid and a preimage of the output. For this to work, the signature must be *unique*, which means that only a single valid signature exists for a given message and key. Due to this requirement and the ease of constructing threshold implementations, BLS signatures are most frequently used for this purpose [41]. VRFs are used most frequently for leader election in proof-of-stake systems, such as Ouroboros Praos, Algorand, and Fantômette (see Sections 13.1.3, 13.4.2, and 13.3.1, respectively).

Finally, *verifiable delay functions* (VDFs) are relatively new primitives, which are functions that require a significant amount of sequential computation but where the correctness of the result is easy to verify. VDFs can be thought of as a time delay that is imposed upon the generation of output for a pseudorandom number generator. The delay prevents malicious actors from influencing the output of the generator because all of the inputs to the generator are finalized before the delay ends. VDFs are used for randomness in leader election, may be helpful against certain attacks on proof-of-stake and proof-of-space systems, and can be used to limit the frequency with which an adversary can send messages or vote in consensus [42]. VDFs improve upon VRFs in that a single honest participant is required instead of a non-colluding honest majority (requiring a single honest participant is sometimes called the *anytrust* model). The idea of VDFs was first formalized in [43], but improved versions were found soon after [44, 45]. Due to being relatively new, there are still some uncertainties regarding their security, including the possibility of specialized hardware and tuning the time parameter properly.

The VDF from [44] works by choosing a time parameter T, a finite abelian group G of unknown order, and a hash function H with a domain that consists of the elements of G. For an input x, let $g = H(x)$. The VDF is evaluated by computing $y = g^{2^T}$. Repeated squaring does not reveal any information about the output until the final squaring, and the computation must be done in serial (because the order of G is unknown). Unfortunately, generating a group of unknown order requires a trusted setup.

2.8. State Machine

The state machine is the set of rules that replicas enforce while transitioning the state of the system via client-submitted transactions. Though introduced here, Section 18 provides a more detailed discussion of the design space for state machines.

A wide variety of rules are possible, but they typically include ensuring that transactions have valid signatures of the clients who are authorized to act on the portion of the state involved, as well as preventing "double-spending." In the context of a cryptocurrency, this means that the owner of the coins signed the transaction and that there is not a conflicting transaction that spends the same coins included in the ledger. Other rules may include a maximum allowed block size, restrictions on the timestamps in block headers, that blocks and transactions are syntactically well-formed, and that all transactions in a block are committed to in the block header.

In addition to these basic rules, the state machine may include a means of executing various types of computations, which are encoded as *smart contracts*. The state machine often provides a programming language and execution environment where developers can write programs (smart contracts) that are executed by all of the network's replicas, who then agree on the result and modify the system's state accordingly. In some systems, such as Ethereum, the execution environment is essentially *Turing-complete*, so it can execute any arbitrary deterministic program subject to the system's *block gas limit*. Each operation in the state machine has an associated *gas cost*, where *gas* is a unit of measurement that corresponds to the amount of effort or resources consumed to execute the operation. The gas limit is analogous to a maximum block size, but instead of limiting the total size of transactions, it bounds the total computational effort.

2.8.1. UTXO vs. Account Model

There are two common models of how the state machine's state is represented: the *UTXO model* and the *account model*. A UTXO, short for *unspent transaction output*, is an immutable object associated with a particular spending condition. By fulfilling this spending condition, a client becomes authorized to destroy the UTXO and create new ones from it (so long as other state machine rules are followed, like not creating more new coins than were spent). In the UTXO model, each transaction takes some UTXOs as inputs, destroys them completely, and creates new UTXOs as output with different spending conditions that reflect the new owner. This often includes "change" outputs that return funds to the sender.

In contrast, accounts are mutable objects associated with a balance that can increase or decrease as funds are moved in and out of the account. The UTXO model creates a directed graph of transaction outputs that move between owners (shown in Figure 3), whereas the account model is like a database of the current system state. One can think of UTXOs as individual dollar bills with arbitrary denominations, and accounts are more like regular bank accounts with an identifier and a balance. In either case, the UTXO or account may have programmable spending conditions.

Each model has particular advantages and disadvantages. In the UTXO model, transactions may be more private due to the ease of creating new addresses for each output, whereas all of the transactions to and from a particular account in the account model are automatically linked. Due to the immutability of UTXOs, disk access to check system state is more par-

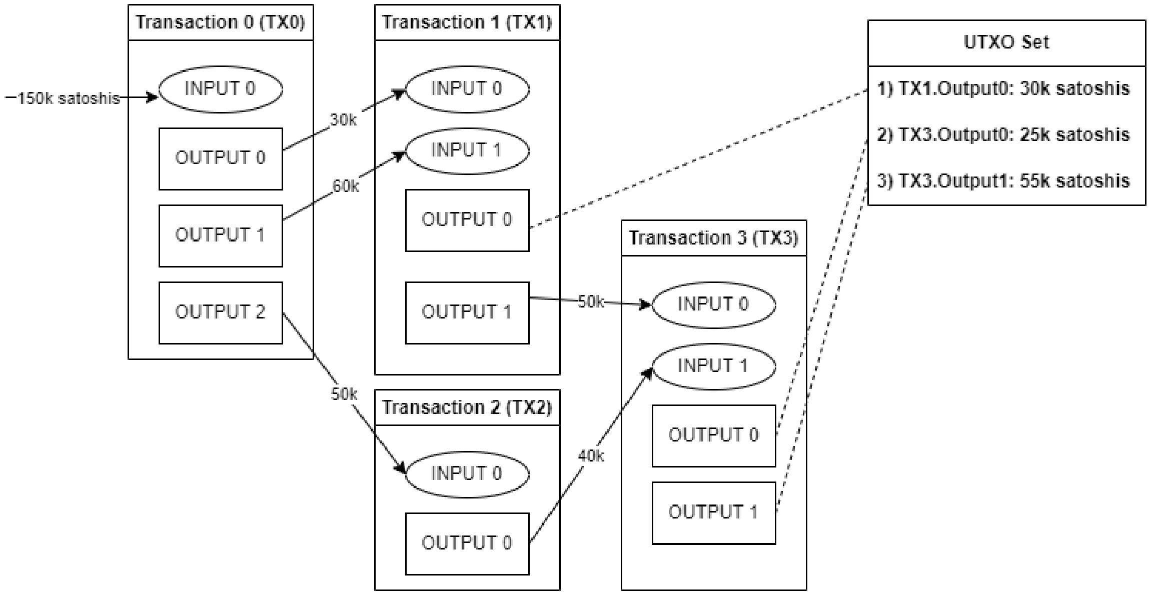


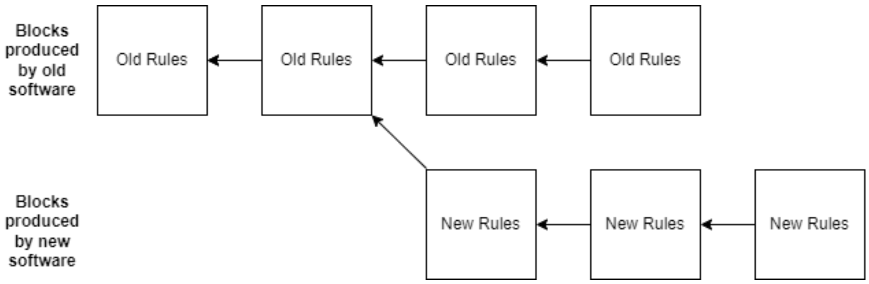
Fig. 3. UTXO transaction graph. Outputs of some transactions become inputs to later ones. Each transaction depicted includes a 10k satoshi fee, where satoshis are the smallest atomic unit of currency in the Bitcoin network.

allelizable than in the account model, where the accounts can be modified. On the other hand, the account model has the performance advantage of typically smaller transactions. In the account model, every transaction has a single "input" and "output." With UTXOs, some transactions may become extremely large if there are many recipients or if they require simultaneously spending many UTXOs that have a small value. The account model is also simpler for light clients; in the UTXO model, a light client must keep track of each UTXO owned by the client and update this for every transaction. One of the biggest disadvantages faced by the account model is that transactions require nonces in order to prevent replay attacks. The nonces must be sequential, which can impact transaction processing if an account issues many transactions in short periods of time (they must be processed sequentially, regardless of network lag and the attached fees).

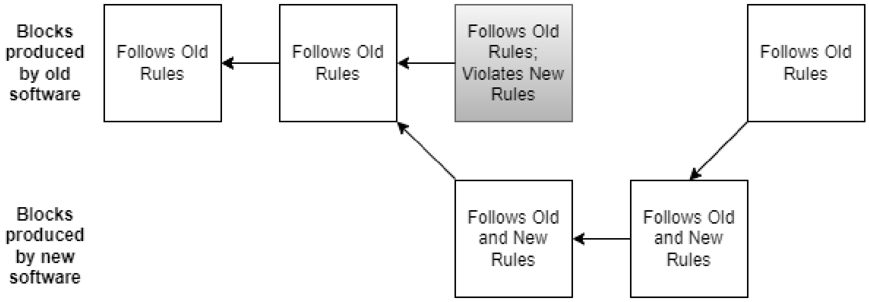
Perhaps the most straightforward advantage of the account model is that accounts are able to maintain a persistent state, which can make complex smart contract programming much simpler than in the UTXO model, where UTXOs do not maintain state. The statelessness of UTXOs makes it challenging to program smart contracts with multiple phases. In the account model, a smart contract exists at a static address, which can store state and be referenced without updating.

2.8.2. Changing the Rules

All validators must agree on the state machine rules. However, the rules themselves may change over time. Broadly speaking, there are two types of rule changes, or *forks*: *hard*



(a) Hard Fork



(b) Soft Fork

Fig. 4. Hard forks and soft forks.

forks, which make blocks or transactions valid when they were previously considered invalid, and *soft forks*, which make blocks or transactions invalid when they were previously considered valid. Hard forks are not backward compatible, whereas soft forks are. The different fork types also have different security ramifications, different practical considerations regarding the mechanics of implementing the fork, and – according to some – different ethical considerations. Hard forks are depicted in Figure 4a, and soft forks are shown in Figure 4b.

Hard forks can be thought of as the creation of an entirely new system, where a successful hard fork results in a smooth transition from the old system to the new system. Technically, a hard fork always results in two separate systems, but a smooth hard fork with (near) universal consent from the community may result in the original chain being abandoned. Nodes that fail to update their software may be more easily attacked around the time of the hard fork, so hard forks require near-unanimous consent and for users to all update their software within a specific period of time. For example, a node that has not upgraded can be presented with a chain of blocks under the old rules by an attacker. Since those blocks will not be reorged, the attacker can trick the recipient into accepting coins that the remainder of the network will not accept. In Figure 4a, a node that follows the old rules and accepts transactions with only two confirmations can be easily defrauded.

In contrast, soft forks tend to be more secure than hard forks because they do not require every user to update their software near-simultaneously, although those actively participating in consensus, such as miners and staking validators, should update prior to the soft fork activating. For a soft fork to happen smoothly, the majority of the mining power (or stake) should recognize and enforce the soft fork; a supermajority would be ideal. If some miners are running software that enforces the old rules, users could receive "fake" confirmations in blocks that are valid under the old rules but invalid under the new rules and, thus, soon to be reorged. This is the scenario created by the shaded block in Figure 4b; if a user's node follows the old rules and accepts transactions with a single confirmation, they can be defrauded. As a result, non-upgraded nodes lose some security during a soft fork.

Both hard forks and soft forks can potentially result in permanent chain splits, but it is far less likely to occur with soft forks and can be avoided completely through the choice of activation mechanism. Permanent chain splits create considerable complications for users. First, the values that accrue from network effects are reduced by splitting the relevant communities in two. Second, unless the fork changes the rules for how transactions are constructed, some transactions that are valid on one chain will also be valid on the other in case of a split, which creates the opportunity for *replay attacks*. An attacker can take a transaction from chain A, broadcast it again to the network maintaining chain B, and have it confirmed. If a chain split occurs and a cryptocurrency exchange only supports chain A, then an attacker who initiates a withdrawal from the exchange on chain A can copy the transaction, broadcast it over the network managing chain B, and receive an equivalent amount of currency B that the exchange never intended to give up.

Chain splits cause problems for blockchain service providers, such as wallets and exchanges. Exchanges must decide whether to support one or both sides of the split, and users of the exchange may not have the latitude to choose which chain to follow. As a result, a user who wanted to stay on the old chain but used an exchange that only follows the new chain could potentially lose their funds. Chain splits also pose a problem for light clients. Because light clients do not validate the state transitions of the network, they are unaware of what rules are being followed. That means that they will simply follow the majority, which can be dangerous. For example, if a majority of the Bitcoin hash rate decided to increase inflation, full nodes would recognize the block as invalid, but light clients would simply follow the majority of the hash rate. In addition to the explicit danger of attacks during this period, if a significant portion of the community were to engage in commerce on the attacking/inflation chain, then a social consensus may form around the chain despite its invalidity. Contentious hard forks that are likely to result in a split are especially dangerous to light clients, which make up the bulk of typical users. More information on light client security compared to fully verifying nodes can be found in Section 3.2.

There are several methods for performing forks that primarily depend on whether they are user-activated or miner-activated. In a *user-activated soft fork* (UASF), the software implementation begins enforcing the new, more restrictive rules at an agreed-upon time, or "flag day." In a *miner-activated soft fork* (MASF), the new rules are enforced only after an extended period of miner signaling. Miners can set a particular bit in their block headers if they support the rule change, and after a threshold percentage of blocks (usually 75%

to 95%) within a given time period signal support, the new rules can be enforced (usually after a short delay to give time for more nodes to update their software once the change is "locked in"). Only a UASF can cause a chain split, but MASFs can be chaotic and result in many short forks if the threshold is set too low. One issue with miner activation is that miner signaling is in no way a guarantee that those miners will actually enforce the new rules. For example, the Bitcoin community debated on whether to increase the maximum block size, and supporters of an increase released a (hard fork) client called Bitcoin XT that enforced a block size increase, while the dominant client – Bitcoin Core – did not. An anonymous developer then released the NoXT client, which mimicked Bitcoin XT's behavior and signaling but actually followed rules compatible with the Bitcoin Core software [46]. If some miners were running this client, the fork would be activated prematurely to the detriment of those who wanted to enforce the new rules.

User activation has its own problems. The idea of a UASF is that if the majority of economically significant users of the system adopt it, then the miners will be forced by their own self-interest to adopt it as well. As a result, the UASF may be able to get around the resistance of miners if they are acting in opposition to the broader community of users. Again, the primary issue is that it is likely to result in a chain split without the support of a majority of the miners. In fact, without majority mining support, a UASF looks a lot like a hard fork with all of the risks that hard forks entail, all of the limitations of soft forks, and the possible need for a hard fork to protect against the miners that do not accept the fork rules [47].

For any fork to be successful, nodes need to coordinate with each other regarding both the time to lock in as well as – after a delay – the time to activate the rule change. In Bitcoin and similar systems, there are several notions of time that can be used for this coordination: the block timestamp, the *median time past* (MTP), and the block height. The block timestamp is unsuitable for use because it does not increase monotonically, which is a requirement for coordination. The MTP is defined as the median timestamp of the block and its 10 predecessors. Bitcoin has rules about the timestamps that force MTP to monotonically increase though it is not monotonic in the face of reorgs. In all but the most exceptional of circumstances, block height will be monotonic even across reorgs. The advantage of using MTP to coordinate is that the timing can be selected in order to minimize the chance that the fork occurs at particular times. This may be useful in order to time the fork to happen while, say, engineers at important companies may be at work instead of asleep. On the other hand, MTP can be manipulated by miners in a number of ways through their control of the block timestamps, and a majority of miners can use this mechanism to completely skip over relevant activation periods [48]. Block heights are safe from this risk but are less likely to match a desired time for the humans running the nodes or the developers writing the code that schedules activation.

Many in the blockchain community believe that there are ethical considerations with regard to the style of fork employed. This goes beyond the nature of the specific rule changes themselves (e.g., a rule that allows a privileged party to steal from others) but rather to the meta-problem of the methodology of forking. Intuitively, in a distributed network composed of many heterogeneous individuals with potentially competing interests, any potential change in the rules (other than, say, a fix for a catastrophic software bug) is likely to be opposed by at least one person. As a result, if the change occurs, there may be a perception that this person or people were coerced into accepting the change. A representative of this view is Ethereum creator Vitalik Buterin, who believes that both types of forks are coercive, but that soft forks are more coercive than hard forks:

> There is an essential difference between hard forks and soft forks: hard forks are opt-in, whereas soft forks allow users no "opting" at all. In order for a user to join a hard forked chain, they must personally install the software package that implements the fork rules, and the set of users that disagrees with a rule change even more strongly than they value network effects can theoretically simply stay on the old chain...In the case of soft forks, however, if the fork succeeds the unforked chain does not exist. Hence, soft forks clearly institutionally favor coercion over secession, whereas hard forks have the opposite bias...If I had to guess why, despite these arguments, soft forks are often billed as "less coercive" than hard forks, I would say that it is because it feels like a hard fork "forces" the user into installing a software update, whereas with a soft fork users do not "have" to do anything at all. However, this intuition is misguided: what matters is not whether or not individual users have to perform the simple bureaucratic step of clicking a "download" button, but rather whether or not the user is coerced into accepting a change in protocol rules that they would rather not accept. And by this metric, as mentioned above, both kinds of forks are ultimately coercive, and it is hard forks that come out as being somewhat better at preserving user freedom. [49]

As alluded to by Vitalik, others take the position that hard forks are more coercive or at least more problematic from the standpoint of the developers who must write code that implements the hard fork. While most users agree that hard forks may be necessary in some cases, some believe that *controversial* hard forks should be avoided. For example, Bitcoin Core developer Pieter Wuille, in the context of a debate about changing the maximum block size in Bitcoin, expressed a distaste for hard forks that lack near-universal agreement in the Bitcoin community:

> [The responsibilities of the Bitcoin Core developers] include[] participating in discussions about consensus changes, but not the responsibility to decide on them – only to implement them when agreed upon. It would be irresponsible and dangerous to the network and thus the users of the software to risk forks, or to take a leading role in pushing dramatic changes. Bitcoin Core developers obviously have the ability to make any changes to the codebase or its releases, but it is still up to the community to choose to run that code...Bitcoin Core is not running the Bitcoin economy, and its developers have no authority to set its rules...Worse, intervening in consensus changes would make the ecosystem

> more dependent on the group taking that decision, not less. So to point out what I consider obvious: if Bitcoin requires central control over its rules by a group of developers, it is completely uninteresting to me. Consensus changes should be done using consensus, and the default in case of controversy is no change. [50]

Essentially, this view suggests that it is not the role of developers to decide what Bitcoin is or should be and that developers have a responsibility to not "force" controversial changes to the system. To do so would elevate the developers to a more powerful position than they ought to have and would go against the decentralized ethos of the community.

Over the lifetime of a long-lived system, it is likely that a hard fork will be necessary, and soft forks are simply not preventable when supported by a majority of miners so perhaps the ethics are less relevant than practical concerns. In either case, no users are forced into running particular code on their machine. Ultimately, users decide the rules because users decide which software to run. The "coercion" in either case is in the eyes of the beholder – no one is forced to use any particular rules, but they may be "forced" to use particular rules if they want to remain compatible with their counterparties.

3. Scaling and "Decentralization"

3.1. A Note on Decentralization

The term *decentralization* is used frequently when discussing modern replicated state machines. However, there is no single, accepted definition of what decentralization means. As a result, its use often confounds more than clarifies. It is worth investigating a few of the many proposed definitions of decentralization in order to grasp the diversity of views.

Balaji Srinivasan and Leland Lee propose a metric they call the *(minimum) Nakamoto Coefficient* to measure the decentralization of a system [51]. The idea behind this metric is to first create a list of the essential subsystems of the decentralized system under analysis, determine the number of entities that an adversary would need to compromise in order to take effective control over each subsystem, and take the minimum as a measure of the system's decentralization. The system is more decentralized when more entities must be corrupted to control an essential subsystem.

There are many plausible subsystems, though it is unclear which ought to be considered "essential." Subsystems may include the distribution of mining rewards, the number of exchanges, the volume traded on exchanges, the number of software clients, the number of developers per client, the number of full nodes being run (and their distribution by legal jurisdiction), the distribution of asset ownership, the fraction of users that hold their own private keys instead of delegating to a custodian, the fraction of users who validate the ledger with their own node instead of trusting another entity, the number of businesses running economically significant nodes (i.e., who validate many incoming transactions), the number and distribution of hardware manufacturers in proof-of-work systems, the number of mining pools, and so on.

In addition to the challenge of determining which of these subsystems are essential, it can be difficult to determine what the proper threshold of corrupted entities would need to be for some subsystems. For instance, it is unclear what percentage of exchange volume would suffice to consider that subsystem captured or centralized. In addition, reasonable people may disagree on the benefits of having multiple clients instead of a single client with more developer attention. In other cases, the subsystems may take on a different significance depending on the network in question. For example, the concentration of coin ownership is more important in a proof-of-stake network than in a proof-of-work network. If the wealth within a system is highly concentrated, then a government or other powerful adversary may only need to target a few large holders in order to acquire a large enough fraction of the asset and cause a market crash.

Paul Sztork proposed an alternative measure of decentralization: the *cost of node-option* (CONOP) [52]. He points out that "[t]he process of 'money' is 'knowing you've been paid'" and that a process is more decentralized when it happens more locally. As a result, he says that "'decentralized money' is the local cost of knowing you've been paid: the cost of running a full node." This can be generalized to arbitrary computation possible in state machine replication: decentralized computation can be measured by the cost required to validate the correctness of computations. Actually running a fully validating node is not strictly necessary for everyone. Instead, the relevant consideration is the cost required to start and operate a node should they desire to do so. Ultimately, CONOP is intuitive because if it is extremely easy to set up and run a full node, the network remains difficult to shut down or prevent access to, even if there are not many nodes running at any given time. On the other hand, if only a handful of entities have the resources to run a fully validating node, the system can easily be shut down or succumb to undue influence. In this case, most individuals would be required to trust a third party and would not be able to locally verify for themselves whether the payment or computation happened correctly.

A final decentralization metric proposed in [53] is (m, ε, δ)-decentralization. This is defined as a state of the system such that at least m participants run nodes that participate in consensus, and "the ratio between the total resource power of nodes run by the richest and δ-th percentile participants is less than or equal to $1 + \varepsilon$" [53]. Since this metric is only concerned with the centralization of the Sybil-resistance aspect of the protocol, it is more constrained in scope than the Nakamoto Coefficient or CONOP. The intuition for this metric is that it is preferable to have a large number of participants involved in consensus, and the participants ought to have a roughly even distribution of power. The paper derives four necessary conditions for a system to have "full decentralization" according to the authors' conception of the term:

1. Any nodes with resource power must earn rewards (and there must be at least m of them).

2. Participants should find it more profitable to run their own nodes than to delegate their resources to another participant.
3. It should not be more profitable for an entity to run several nodes than to run a single one.
4. "[T]he ratio between the resource power of the richest and δ-th percentile nodes converges in probability to a value of less than $1+\varepsilon$" [53]. That is, the advantage of the most well-resourced node is bounded.

Crucially, the third condition is not possible without a trusted third party, leading to the paper's conclusion that "full decentralization" is impossible in permissionless systems. Without some form of trusted identification authority, it is unclear how to design a system where it costs more for one entity to run multiple nodes than it costs for multiple individuals to each run their own node.

Each of these conceptions of decentralization highlights a different but important aspect of system security and architecture. The Nakamoto Coefficient highlights the fault tolerance of the system in an intuitive way: a system is more decentralized when it can tolerate a greater number of faulty or malicious entities. Practical challenges to using this as a metric include a lack of clarity regarding which subsystems should be included and the resultant difficulty of comparing different systems when their component subsystems differ in importance. In contrast, CONOP has nothing to do with fault tolerance. Instead, CONOP highlights the barriers to entry for users to be able to take advantage of the security properties of the distributed system. In this conception, a more decentralized system is one where a greater number of users have the ability to fully validate the replicated state machine – that is, to be an *equal* peer in a peer-to-peer network rather than dependent on an external trusted party. Finally, (m,ε,δ)-decentralization highlights the impact that wealth or resource inequalities outside of the system have on the relative power of participants within the system. In an open, permissionless system, these inequalities are likely to be reflected in the distribution of the Sybil-resistance resource. In this case, a system is more decentralized when the resource disparities between more powerful nodes and less powerful ones are smaller.

In this document (NIST IR 8460), the term "decentralization" is used sometimes but sparingly. It does not refer to any specific meaning above but should be clear based on the context in which it is written.

3.2. Full Nodes and Light Clients

There are multiple distinct operating modes that a user might choose when interacting with a replicated state machine. These modes can be broken into two categories: *full nodes*, which process and validate the complete historical ledger of transactions, and *light clients*, which may validate a portion of the ledger or a subset of the rules but require placing some amount of trust in other entities.

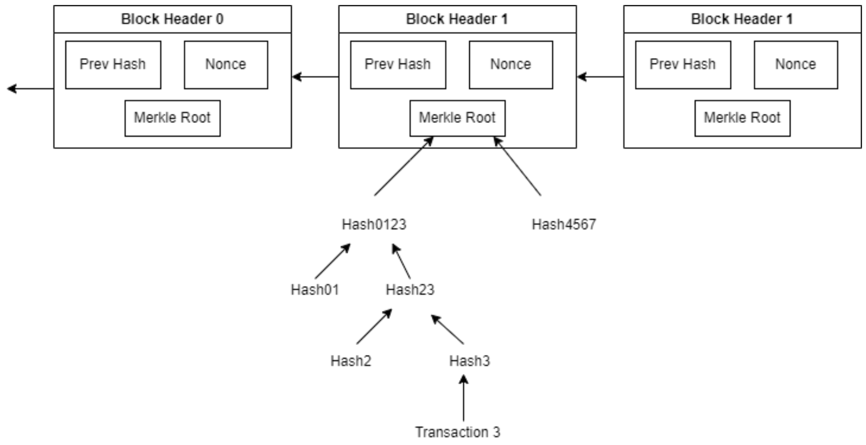


Fig. 5. Simplified Payment Verification (SPV). A light client with a chain of block headers can verify the inclusion of Transaction 3 in Block 1 when provided with $Hash2$, $Hash01$, and $Hash4567$.

There are different types of light clients, but the original one proposed in the Bitcoin white paper uses *Simplified Payment Verification* (SPV) [1]. SPV clients connect to full nodes but do not download the entire blockchain. Instead, they only maintain the chain of block headers for the best chain they are aware of (in Bitcoin and similar systems, this is the chain with the most cumulative proof of work). Because the block header contains a Merkle root that commits to every transaction contained in the corresponding block, it is possible to create proofs of transaction inclusion that scale logarithmically with the number of transactions in the block. This is shown in Figure 5. While this process does not demonstrate that the transaction is valid, it proves that the transaction is included in the canonical chain if connected to at least one honest full node. This implies that the transaction can be trusted to be executed and not reverted if the adversary is beneath the security threshold of the system and the block has enough confirmations. Alternatively, it shows how much work would be required to revert the transaction and perform a double-spend.

The full nodes that serve the SPV client need to know which transactions the client cares about in order to provide a Merkle inclusion proof for it. To this end, the SPV client constructs a *Bloom filter* – a compact, probabilistic data structure that allows one to test whether an item is a member of a set – that includes the receiving addresses that the user wants monitored. The Bloom filter includes false positives, such that some of the addresses are not ones that the client cares about. In theory, this should substantially improve privacy for SPV users compared to simply providing a list of addresses. In practice, however, Bloom filters provide almost no additional privacy for light clients, and users should understand that they are – in essence – telling a service provider about all of their transactions (and possibly associating them with an IP address) [54]. Perhaps for this reason, many light clients actually do simply provide a list of addresses to a wallet service provider and completely sacrifice transaction privacy.

An alternative model of light client improves privacy at the cost of significantly increased bandwidth (though still substantially less than a full node). *Compact block filters* are compressed representations of the contents of blocks using Golomb-Rice coding and allow a light client to check whether the block contains transactions that are relevant to the client [55, 56]. If so, the light client downloads the full block in order to extract the relevant transaction. In addition to downloading a subset of complete blocks, the filters impose greater resource requirements than SPV but scale better for the full nodes that serve data. For each block, a full node only needs to create the filter once and can send the same filter to any light client who asks. In contrast, each SPV user provides its own Bloom filter, and the full node needs to perform considerable disk I/O to check for transactions for each of them.

Regardless of the method used, light clients have degraded security properties compared to full nodes, which is the price paid for convenience and low resource usage. First, as mentioned previously, light clients reduce user privacy compared to full nodes. Even compact block filters reveal the subset of blocks in which the user is the recipient of a transaction. Second, nothing prevents the full node server from lying by omission and failing to tell the light client that they have received a transaction. While this can be mitigated by connecting to multiple full nodes, many wallets connect to only a single server, and there is a risk that the full nodes are coordinating a Sybil attack and are not independent. Light clients need to trust that there are available sockets to connect to these honest full nodes, which full nodes lack an incentive to provide and which has not always been the case in practice [57]. Compact block filters also mitigate the problem of lying by omission by allowing the light client to check the validity of the filter itself to see if the full node is lying. Third, light clients are more susceptible to certain attacks, like the vector76 attack described in Section 10.3.

More importantly, light clients have no way of knowing whether the chain that they are following is valid in the first place. Light clients do not maintain the full system state that would be required to validate transactions. As a result, light clients can be deceived by adversaries in ways that full nodes cannot. The only thing that light clients have assurance on – assuming that they are connected to at least one honest full node – is that they are aware of the chain that satisfies the network's chain selection rule, ignoring the requirement that the chain be valid. As a result, light clients can be deceived into trusting an invalid state if the security threshold of the system is violated temporarily, if the adversary does not exceed the threshold but gets lucky, or if the adversary can partition the victim from the rest of the network. As a result, a miner can double spend a limitless amount of funds against a light client, whereas they can only double spend what they already own against a full node. Similarly, light clients do not recognize hard forks if the fork does not change the semantics of the block header and may, therefore, be "forced" to accept a rule change that the user of the light client would not otherwise approve of.

A possible solution to this issue is *fraud proofs*, which are "alerts" that a full node can provide to a light client to convince the light client that a block is invalid. There are many subtle challenges to implementing fraud proofs, which are elaborated on in Section 15.5.

However, even with a working fraud proof implementation, an adversary may still be able to partition the light client, filter out fraud proof messages, and launch the same attacks, which would not work against a full node.

In short, while running a full node may be dramatically more resource-intensive than a light client, there are clear benefits. This includes substantially better privacy and requiring far less trust in third parties to be honest about the state of the ledger.

3.3. Scalability Challenges and Block Sizes

One of the most frequently discussed issues in the blockchain community is the scalability of the underlying technology. In particular, debates about the appropriate means to scale the Bitcoin network led to the high-profile August 2017 hard fork that created a new asset – Bitcoin Cash – with a larger maximum block size than Bitcoin proper. New replicated state machine projects routinely advertise themselves as being highly scalable. Usually, the claim is that a project can handle more transactions per second (TPS) than its competitors.

Scalability in the context of replicated state machines can mean several different things. As mentioned above, it is often used in reference to increasing the system's maximum transaction throughput, which is usually measured in TPS. Another scalability metric is the latency of transaction commitment, where a more scalable system has lower latency. Finally, it can also be used in reference to the total number of validators able to participate in consensus, such that a more scalable system is one that can function with a greater number of participating validators.

Some of the confusion regarding scalability is a result of these different meanings of the term. This problem is greatest when comparing a permissioned system to a permissionless one based on TPS but can confuse a comparison between two permissionless systems as well. Despite being common, these comparisons are unscientific. In a permissioned network, one can fix the number of validators – n – of which f are byzantine, define a standard transaction, measure TPS in a controlled environment, and make comparisons *to other permissioned systems* while holding the number of validators constant (as well as other environmental factors). In a permissionless network, n naturally fluctuates. Furthermore, it is often the case that systems that scale to a larger number of validators are not capable of tolerating as high a level of throughput. That is, there is a trade-off between maximizing n and TPS, and increasing n is just as legitimate of a scaling goal as increasing TPS. Therefore, TPS does not provide a scientific way of comparing permissionless networks where n is not constant.

There is no single, correct answer to how many consensus-participating nodes a system should have, and the trade-offs between types of scalability imply that this is highly subjective. The answer depends on the goals of the system designer, the system's architecture, and the preferences of the system's users. In a permissioned system, the existence of more replicas implies that more organizations can participate in the consortium, so the relevant number likely relates to the structure of the industry in question. Because the replicas are run by organizations that may be able to afford expensive hardware and bandwidth, permissioned systems can focus more on maintaining high TPS while assuming that substantial resources are provisioned at each node.

Permissionless systems are more complicated, in part because it is common to have fully validating nodes that do not directly participate in consensus (i.e., by mining or staking). In a certain sense, the only node that one needs to care about is their own as long as the security threshold of the system is not violated. However, the network itself likely requires hundreds of nodes to provide sufficient diversity in terms of legal jurisdiction and geographic region to make it challenging for a moderately powerful adversary to disrupt the network. At the same time, many thousands of nodes are likely needed in order to serve the population of light clients and make the network useful to a large number of people. In permissionless networks, the intention is to allow anyone to participate anonymously. As such, it must target users with more average-to-worst-case hardware and bandwidth.

Running a full node requires a number of computing resources, some of which present considerable scaling challenges. Consider the job of a full node in a typical blockchain system, such as Bitcoin. To join the network, a new node must download the entire blockchain history all the way back to the genesis block. Then the entire chain must be fully validated (although there are some possible shortcuts that can reduce security in order to improve initial synchronization time) to build and evolve the system state as it goes. This usually involves checking that at least one digital signature per transaction, as well as checking that it is consistent with the system state, which is held either in memory or on disk. Upon synchronizing with the network, the node must constantly verify transactions and blocks as they are gossiped over the network while relaying this data over several network connections as quickly as possible.

The historical blockchain can be stored on a hard disk drive (HDD) because what occurred in the past does not need to be accessed frequently. However, enough nodes need to maintain this data in order to serve it to new nodes who join the network and want to perform full validation. The history does not require quick access, but the state of the system does (in Bitcoin, this is the UTXO set). In order to validate transactions and blocks as quickly as possible, the state should be stored entirely in system memory. When this is not possible, some of the state must reside in more persistent storage. However, frequently accessing this state can result in a significant disk I/O burden. For example, Ethereum's system state is tens of gigabytes, making it all but impossible to run a node without a solid state drive (SSD), which provides quick random access to the system state. A node using an HDD is unlikely to ever remain in sync with the rest of the network. Unfortunately, RAM is expensive, and the state size can – in the worst case – expand as quickly as the maximum block size. To verify transactions, a node will heavily utilize a CPU. Even the simplest transactions almost always require verifying a digital signature, which is a fairly expensive operation. Worse, specially constructed transactions can be extremely time-consuming to validate (see Section 19.2 for a more detailed description of this issue), and in permissionless systems, it is necessary to consider worst-case algorithmic complexity to avoid attacks. Finally, this all takes place over a network, and sufficient networking resources must be deployed in order to remain in consensus. Bandwidth is important for all nodes (upstream bandwidth in particular), and latency is critically important for miners. The node must be connected to several other full nodes to exchange data and may also need connection slots and other resources to serve light clients.

Perhaps the greatest scaling challenge is with the initial blockchain download and synchronization (IBD), which inherently becomes a bigger and bigger problem over time. In Bitcoin, IBD could well be impractical today were it not for a custom cryptographic library – *libsecp256k1* – that verifies signatures dramatically faster than OpenSSL, which was used before [58]. After exhausting the low-hanging fruits of performance improvements, IBD performance will likely take longer and longer. This can create a potential DevOps problem, as it is typical to resynchronize a node any time validation code changes. If it takes a month to synchronize, development will be significantly hampered. Far worse, if IBD becomes impossible or sufficiently challenging, the free entry condition of permissionless networks may be violated.

Performance is especially important for miners and, more broadly, those who participate in any synchronous consensus protocol. The latency of block propagation is a major contributor to the centralization of power among consensus-participating nodes and is discussed in detail in Section 10.2.1. In short, larger miners are more likely to win in block "races" when two miners mine blocks at around the same time. With higher latency – such as that caused by larger blocks – more forks will occur, providing the larger miner with disproportionate rewards over time. Some argue that this is acceptable or even desirable and represents a healthy competition that results in miners upgrading their hardware and provisioning greater bandwidth access, which can lead to an increase in TPS. Bitcoin Core contributor Peter Todd responded to this by pointing out that requiring miners to dedicate more resources to survive actually reduces decentralization and makes those TPS worth less:

> What’s tricky is designing a Bitcoin protocol that creates the appropriate incentives for mining to remain decentralized, so we get good value for the large amount of money being sent to miners. I’ve often likened this task to building a robot to go to the grocery store to buy milk for you. If that robot doesn’t have a nose, before long store owners are going to realise it can’t tell the difference between unspoilt and spoilt milk, and you’re going to get ripped off paying for a bunch of spoiled milk. Designing a Bitcoin protocol where we expect "competition" to result in smaller miners in more geographically decentralized places to get outcompeted by larger miners who are more geographically centralized gets us bad value for our money. Sure it’s "self-correcting", but not in a way that we want. [59]

Block propagation latency is not the only incentives-related factor that can lead to centralization in the absence of a sufficiently small maximum block size. For instance, in a system with no single hard limit on the maximum block size (where miners communicate their own desired limits instead), consensus becomes unstable and can lead to lengthy forks that benefit miners of larger blocks at the expense of those who can only process smaller blocks [60]. More to the point, miners who are able to handle larger blocks can form a "cartel" of sorts that increases the block size and makes smaller miners unprofitable [60]. A similar cartel-forming result is likely to hold if the maximum block size is sufficiently high, even if there is agreement on it. The break-even cost for a miner to include a transaction in a block decreases as the mining pool size increases because orphan risk decreases. As a result, larger pools will be able to process more transactions and collect more transaction fees from users, forcing smaller miners out of business as the mining difficulty adjusts.

Another major problem with increasing the on-chain throughput of a permissionless system is that miners become more and more incentivized to completely skip validation as the burden of validation increases. Miners would prefer to just assume that a block they receive is valid and start mining on top of it, which will provide a profitability advantage by starting work on the next block more quickly. This is especially significant if transaction fees are not a major portion of the miner reward, which is more likely to be the case with larger block sizes (an increased supply of block space leads to a decrease in the price of transaction inclusion). Unfortunately, this can result in (portions of) the network accepting invalid blocks, which happened on the Bitcoin network on July 4th, 2015 [61]. Six blocks were built on top of an invalid block because enough mining pools failed to validate blocks they received at the time. While full nodes merely experienced a slowdown in the growth of the honest chain, light clients could have been subject to attacks during that period, depending on which nodes they were connected to. If mining nodes had not been the only ones misconfigured or if non-mining full nodes were not the dominant type of node on the network, the situation could have been catastrophic, as light clients would have followed the invalid chain by default.

A potential solution to this problem is the idea of fraud proofs, which was introduced in the previous section (and detailed in Section 15.5). If there were a way for full nodes to send an alert to light clients that proved that a block was invalid, light clients could be prevented from following an invalid chain. Unfortunately, even if fraud proofs could be easily deployed, they would not be a panacea. In order to construct a fraud proof, a full node needs access to the data that proves fraud. However, a malicious miner can construct an invalid block with an otherwise valid block header and simply refuse to publish the fraudulent part of the block. Until the fraudulent data is made available, light clients will follow the invalid chain if they are aware of it. In addition, light clients need to actually receive and validate the fraud proof, but this may be prevented by a Sybil attack. If the network is sufficiently centralized or light clients do not connect to many different full nodes, an explicit Sybil attack may not even be necessary.

While fraud proofs would no doubt be beneficial, it is concerning to consider what might happen socially if fraud is not immediately detected, especially in a high-TPS system. If there are not "enough" full nodes (where "enough" is ill-defined), it becomes far more challenging to coordinate a response to fraud. Important ecosystem participants may be able to force undesirable rule changes onto the network and suppress fraud proofs for long enough that substantial commerce may occur on the fraudulent chain. If, say, a day passed before the fraud was noticed, the community would need to roll back a day's worth of honest commerce to correct the fraud, which might be sufficiently damaging on its own that users simply decide to follow the invalid chain anyway. To roll back the chain and continue using the old rules, the community would need to quickly bootstrap a new set of nodes capable of handling the high throughput of the chain. If this task is too challenging, then the fraud is likely to persist.

A possible conclusion that one can draw from this is that the benefits of permissionless systems may fail to hold at sufficiently high TPS. When resource requirements become prohibitive for typical end users, the population of full nodes will decrease and may come to be dominated by a handful of large businesses. These few nodes may end up running on centralized cloud services and create a strong risk of correlated failures that hamper the availability of the system. Market concentration also makes the system more susceptible to censorship or coercive rule changes imposed by external actors, such as governments. As a result, a high-throughput permissionless chain may end up losing the free entry condition described in Section 1.6. With few synchronized nodes, there would be little incentive for them to share the blockchain with new nodes hoping to join the network. In fact, there would be strong reasons for them to avoid doing so: not only would this impose a significant bandwidth cost, but preventing new nodes from synchronizing is an effective way to prevent business competitors from arising. Further, new nodes themselves would have a very challenging time getting synchronized even if current nodes did serve the blockchain. They would need to pay for their own data center provisioning for heavy bandwidth and computation. It is no stretch of the imagination to think that, at this point, identification requirements could be imposed on validators, and the system could devolve into a de facto permissioned network.

Despite all of these risks and problems, the scalability potential for permissionless ledgers is not all bad. Even if block sizes must be constrained, the maximum block size need not remain fixed forever (though a hard fork is required in order to increase it). As the underlying computing resources needed to run a node improve and become cheaper over time, greater throughput becomes possible at the same cost for full validation. In addition, the more (publicly reachable) full nodes there are, the cheaper it becomes to run one. This is because much of the work that a full node performs involves serving others (including IBD), relaying blocks and transactions, and serving light clients. More nodes help spread the burden of these resource-hogging functions. Better networking subprotocols can be deployed in order to reduce the latency of block propagation and its centralizing effects. Some of these protocols are already in use and described in Section 10.2.1. For some applications, scalability can be improved by eschewing a total ordering for transactions, such as with some DAG-based protocols or the reliable broadcast payment schemes mentioned near the end of Section 1.3. Payment channels and state channels can be used to reduce the amount of data stored on-chain and the bandwidth used to distribute such data, as described in Section 18.2.1. These technologies allow transactions and smart contracts to be executed by only the participants involved instead of the entire network. There are a number of ways to further scale computation, such as adding concurrency to the state machine (see Section 18.1.1). Techniques such as optimistic rollups allow most nodes to skip the execution of transactions and rely on rationality assumptions, while zk-rollups allow nodes to skip execution and instead check an easy-to-verify proof of correct execution (see Section 18.2.2). Sharding may be used to distribute the load of computation, bandwidth, and storage among smaller sets of validating nodes, as described in Section 15. Clever use of cryptographic accumulators can reduce the system state to a few kilobytes or less at the cost of increased bandwidth consumption, which may be able to resolve the problem of unbounded state growth. An example of this is the Utreexo proposal, which relies on dynamic hash-based accumulators [62].

The most significant problem is bootstrapping a new node with the IBD process. The system can support a greater amount of activity without imposing a further burden on the initial synchronization process to the extent that many of the above scaling technologies can reduce the demand for block space by keeping transactions off-chain, as is done with payment channels and rollups. Technologies like Utreexo that keep the state size small can improve the synchronization process by allowing it to happen entirely in RAM, obviating the need for slower disk queries. The Mimblewimble protocol, described in Section 18.1, can make the burden of IBD scale with the state size rather than the complete transaction history, which can be a significant difference in practice. More exotic cryptographic constructions like recursive SNARKs can be used to make IBD near instant, as is done in the Mina protocol, which maintains a constant-sized blockchain of less than 22 KB [63].

There are other ways to mitigate the burden of IBD, such as by having either the node software or blockchain include commitments to the system state. While this does not fundamentally solve the IBD scaling problem, it allows new nodes to become useful more quickly, though at a lower security level that is roughly on par with light clients. For example, some Ethereum clients have a "Fast Sync" mode that takes advantage of state commitments contained in block headers [64]. Fast Sync downloads the full blockchain but skips the execution of transactions prior to a specified *launch block*, assuming that transaction execution has been performed correctly up to that point. The node then contacts its peers to request a snapshot of the system state immediately prior to the launch block and verifies that the hash of the state matches the state commitment in the block header at that point. Afterward, the node performs standard IBD from the launch block toward the chain tip and executes transactions normally.

The "assumeutxo" proposal for Bitcoin is similar but with two major differences [65]. First, Bitcoin does not commit to the system state anywhere in the ledger, so a hash of the state is hard-coded into the client software for a block height that is sufficiently far in the past that a significant amount of work has been proven since that block height. The node must acquire the state snapshot itself out of band. Second, while completing the synchronization from the snapshot's block height to the chain tip, a background process starts from the genesis block and executes the complete blockchain up to the assumed valid point to ensure that the state is correct, at which point the security model becomes identical to that of a full node.

4. Practical Byzantine Fault Tolerance (PBFT)

The celebrated Practical Byzantine Fault Tolerance (PBFT) algorithm by Castro and Liskov [4] was the first state machine replication algorithm to possess good enough performance to be used in the real world. Because many other permissioned consensus algorithms have a similar structure to PBFT, this section will describe the system in its entirety. The algorithm is secure under the partially synchronous network model and is optimally resilient, remaining secure against f faulty processes so long as $n \geq 3f+1$. Roughly, the system works as follows:

1. A client makes a request to the leader/primary replica.
2. The primary broadcasts the request to the secondary replicas.
3. Replicas execute the request and reply to the client with the result.
4. The client accepts the result after receiving $f+1$ replies from different replicas with the same result.

In more detail, a client begins with a REQUEST message to the primary with the client ID, the command issued, and a timestamp. When the primary receives the request, they broadcast it and initiate a 3-phase commit (3-PC) process with the rest of the replicas: pre-prepare, prepare, and commit. The first two phases – pre-prepare and prepare – totally order the requests that clients sent in the same view even if the primary replica is faulty. The second two phases – prepare and commit – guarantee that committed requests are totally ordered *across* different views. The normal-case operation of the algorithm can be seen in Figure 6 and works as follows:

1. **Pre-prepare**: The primary broadcasts a signed PRE-PREPARE message with a sequence number, a view number, and a cryptographic digest of the request message. Replicas accept the message if the signature is valid, the digest matches the message, the view is correct, and the sequence number does not match an already accepted sequence number in that view. Because this phase relies solely on the primary and lacks redundancy, packet loss during this phase has the greatest impact on transaction confirmation latency [66].
2. **Prepare**: If a replica accepts a PRE-PREPARE message, it broadcasts a signed PREPARE message to the rest of the replicas, which includes the view number, the sequence number, the request digest, and the replica's ID. Other replicas accept this message if the signature is valid, the sequence number is correct, and the view num-

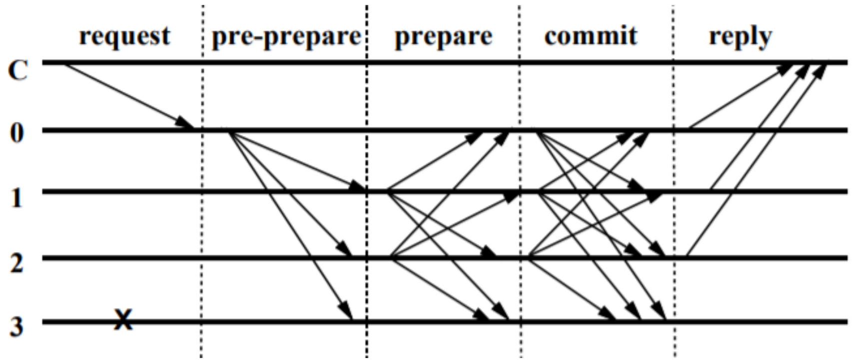


Fig. 6. PBFT normal case operation. In this case, C is the client, replica 0 is the primary, and replica 3 is faulty. [4]

ber matches the replica's current view. A given replica is said to be *prepared* if they have seen a REQUEST, a corresponding PRE-PREPARE, and $2f+1$ PREPARE messages from different secondary replicas that match the PRE-PREPARE (including its own). A replica that is prepared on a given request is also sometimes said to be *locked* on that request. Once locked on a request in a given view, a replica will only vote for that request in later views unless it "unlocks" from the request, which would occur if it finds out that $2f+1$ replicas are not locked on that request in that view or higher. This unlocking occurs when entering the view change subprotocol, as described in Section 4.1.

3. **Commit**: When a replica is prepared, it broadcasts a signed COMMIT message that includes the view and sequence numbers, the replica ID, and the request digest. Replicas accept COMMIT messages if the signature is valid and the additional data matches. A replica is said to be *committed-local* if it is prepared and has accepted $2f+1$ COMMITs from different replicas (including itself) that match the PRE-PREPARE for a given request. A request is said to be *committed* if $f+1$ non-faulty replicas are prepared on that message. The commit phase enforces that if a non-faulty replica is committed-local on a given request, then the request is committed globally. A set of $2f+1$ COMMIT messages from different replicas is sometimes called a *commit certificate* or *quorum certificate*.

Replicas execute the request when they are committed-local on it and when their state reflects the sequential execution of all requests with lower sequence numbers. Replicas respond to the client with a REPLY message with the current view number, the timestamp of the request, the replica ID, and the result of executing the operation. The client accepts after seeing $f+1$ matching replies from distinct replicas. This typical execution of the protocol has communication complexity of $O(n^2)$ when the proposer is honest and the network is synchronous because every replica must communicate with every other replica.

The protocol described above creates an append-only totally ordered log of client-issued transactions that grows without bound. For performance reasons, it would be beneficial if replicas could discard old transactions instead of storing them permanently. Further, if a problem arises that causes a replica to fall out of sync with other replicas, it would be desirable to have a recovery procedure to acquire the missing state or lost messages. A checkpointing subprotocol is used to safely delete old transactions while creating a "proof of correctness" that allows a replica to trust that the state provided during recovery is the agreed-upon state of the remainder of the honest replicas. For safety, replicas cannot delete messages until they know that the associated transactions have been executed by at least $f+1$ honest replicas and that it can prove this during the view-change subprotocol.

A *checkpoint* is generated periodically, such as after a constant number of requests have been executed. When a replica generates a checkpoint, it broadcasts a signed CHECKPOINT message that includes the most recent sequence number and a digest of the state. Replicas store these messages until they receive $2f+1$ CHECKPOINT messages for the same sequence number and digest, at which point the checkpoint is considered *stable*, and these $2f+1$ messages constitute a proof of correctness for the checkpoint. When a replica has a proof of correctness for a checkpoint, the message log for requests up to that sequence number can be discarded. This method involves taking a snapshot of the system state. The system may halt for a few seconds while replicas save their state and stop processing requests. This can be mitigated by having replicas stagger when they take state snapshots [67].

4.1. PBFT View Change

The view change subprotocol provides liveness by allowing the state machine to make progress even when the leader is faulty. At most, liveness can be impeded by f faulty primaries in a row. The subprotocol ensures that replicas agree on the sequence number of requests that commit locally in different views at different replicas.

Secondary replicas start a timer whenever they receive a request (and double the timer length if the view change fails for view $v+1$ before attempting another view change to $v+2$). If the timer expires, the replica initiates a view change and stops accepting messages within the old view. It sends a signed VIEW-CHANGE message with the new view number, the sequence number of the last stable checkpoint and its correctness proof, and the set of valid PRE-PREPARE/PREPARE messages for requests that have not been committed yet in the old view. When the new presumptive primary receives this message from $2f+1$ replicas, it broadcasts a signed NEW-VIEW message with the new view number, the set of valid VIEW-CHANGE messages received, and a set of PRE-PREPARE messages with the new view number. At this point, the primary moves to the new view, and replicas accept the new view if the signature is valid, the view number is correct, and the set of PRE-PREPAREs is valid. The secondary replicas broadcast PREPARE messages for each of these and move into the next view.

If a replica receives $f+1$ valid VIEW-CHANGE messages for views that are not the replica's current view, it will broadcast a VIEW-CHANGE for the lowest view in the set (whether its timer has expired or not) to prevent it from starting a new view change too late.

The communication complexity of the view change subprotocol is $O(n^3)$. The cubic message complexity comes from requiring the new primary to broadcast a NEW-VIEW message with quadratic size that contains $2f+1$ commit certificates, where each commit certificate contains $2f+1$ messages. Because there can be up to f leader failures, even under synchrony, PBFT has worst-case complexity of $O(fn^3)$ or $O(n^4)$. More information on view change protocols can be found in Section 7.6.

4.2. PBFT Security

While rigorous security proofs are out of scope for this document, it is important to understand why algorithms like PBFT are secure, at least informally. PBFT has optimal resilience for a partially synchronous (or asynchronous) protocol, such that the number of replicas $n \geq 3f+1$. If up to f replicas are faulty or have their messages delayed, an honest replica must be able to proceed to the next step of the protocol after having communicated with only $n-f$ replicas. It is possible that those f missing replicas were actually honest, but network asynchrony has delayed their messages. This implies that f out of the $n-f$ communicating replicas may in fact be faulty. Safety requires that a replica must hear from more honest replicas than faulty ones, so that $n-2f > f$, which implies that $n > 3f$.

Many BFT protocols make their security arguments based on *Byzantine quorums* [68] (a generalization of which is discussed in Section 7.5). The idea is that a set of replicas can be divided into a collection of subsets of replicas, called quorums, such that each pair of quorums intersects at a minimum of one honest replica. In theory, each quorum can act independently on behalf of the system, and *quorum intersection* guarantees that operations performed by distinct quorums maintain consistency. To see this, suppose two different transactions at the same position gain $\frac{2n}{3}$ votes (tx_1 and tx_2). Then a set of $\frac{2n}{3}$ distinct replicas (Q_1) voted for tx_1, and another set of distinct replicas (Q_2) voted for tx_2. Then $|Q_1 \cap Q_2| \geq \frac{2n}{3} + \frac{2n}{3} - n = \frac{n}{3}$. By assumption, fewer than $\frac{n}{3}$ replicas are corrupt, so an honest replica is in the set $\{Q_1 \cap Q_2\}$ and voted for both transactions at the same position. However, this is ruled out by the invariant that an honest replica will only vote for one transaction at any given position.

Across multiple views, a locking mechanism provides safety at the end of the prepare phase. If tx_1 is committed in a view, then a quorum of replicas must have locked on tx_1 in that view. If that quorum contains an honest replica that unlocks from tx_1, then another quorum must be claiming to not be locked on tx_1. The intersection of these two quorums contains at least one honest replica, but this honest replica would need to falsely claim that it is not locked on tx_1, which is a contradiction. This demonstrates why 3-PC is necessary instead of 2-PC. A 2-phase commit would fail to achieve safety because a replica cannot guarantee that it will be prepared or locked by a sufficient number of honest replicas. That is, the replica would not know that $f+1$ honest replicas are prepared until receiving $2f+1$

votes in the commit step. Without this assurance, two different requests could be committed at the same sequence number, violating safety. In fact, the "delegated BFT" algorithm used in the NEO blockchain system was essentially a two-phase version of PBFT with safety violations across view changes, and fixing the system involved adding the commit step back in [69, 70].

As discussed earlier, PBFT's liveness is ensured by the view change subprotocol. Liveness can be framed in terms of *quorum availability*, which requires a full quorum of $2f+1$ honest replicas available to respond to an honest leader within a view. Assuming that message delays do not grow exponentially, quorum availability is achieved via three mechanisms:

1. To avoid initiating a view change too quickly, replicas use a timer that grows exponentially with failed attempts at view changes. Exponential growth creates longer and longer periods in which replicas can synchronize their views and achieve quorum availability during a period of network synchrony.
2. To avoid initiating a view change too late, after receiving $f+1$ VIEW-CHANGE messages from other replicas, a replica will broadcast their own VIEW-CHANGE message even if the timer has not run out. This helps bring a lagging replica up to speed more quickly once they know at least one other honest replica wants to change views, and it prevents faulty replicas from forcing view changes too frequently.
3. Faulty leaders cannot indefinitely hinder progress because there can only be faulty leaders for f views in a row at most.

In PBFT, view changes only occur when leaders appear unresponsive or malicious to at least one honest replica. This has an interesting consequence for liveness in that transaction censorship cannot reliably be proven by a client. If a primary refused to order transactions from a particular client (or based on any other criteria), the client would be unable to prove this and force a view change. Therefore, despite formal liveness guarantees, it is possible for a primary to censor requests.

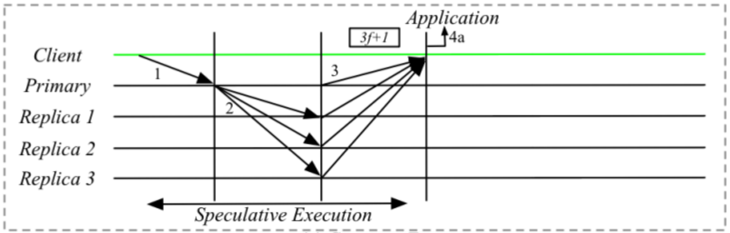
A well-known, open-source PBFT library is BFT-SMaRt, which includes a slight change to the view change algorithm to make the system more modular [71]. Another implementation designed to be compatible with permissioned versions of Ethereum is called Istanbul BFT. However, the original version had both safety and liveness issues [72, 73], which led to an improved "IBFT 2.0" [74] upon being fixed. An alternative fixed IBFT is the one deployed in the Quorum system [75].

4.3. Zyzzyva and Speculative Execution

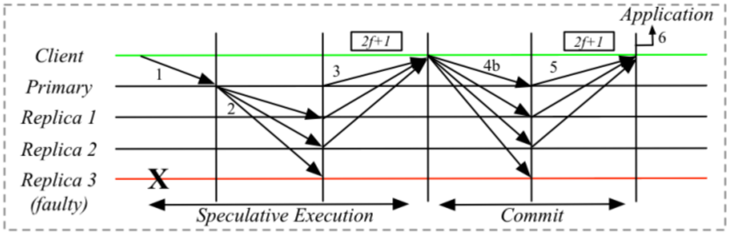
When none of the replicas are faulty, it is possible to significantly improve the performance of BFT protocols. Replicas can execute commands speculatively, assuming that none of the other replicas will experience a fault.

Zyzzyva is an algorithm that pioneered this approach [76]. Replicas (speculatively) execute requests just after receiving the sequence number of the request from the primary and then reply to the client. The client verifies consistency by checking that it received $3f+1$ replies with the same result. At this point, the replicas do not have a guarantee of consistency, but the client does. In Zyzzyva, the client drives the consensus process by informing the replicas if they detect an issue. When a client receives conflicting responses from replicas, the client sends proof (two signed but conflicting messages) to the replicas in order to initiate a view change. The replicas then "roll back" to a safe state prior to the inconsistent execution and change the primary. This approach improves best-case latency and message complexity but risks substantially worse performance when the primary is faulty.

In more detail, the client sets a timer when it makes its request, and if there is a time-out where the client only receives between $2f+1$ and $3f$ consistent replies, it broadcasts a commit certificate to the replicas and starts another timer. Upon receiving $2f+1$ acknowledgements of the commit certificate, the client considers the request complete. If there is a time-out before receiving $2f+1$ local commit acknowledgements, the client broadcasts the request to every replica. The replicas then start a timer, ask the primary to order the request, and initiate a view change if they do not receive word from the primary that the request was ordered. A diagram of the message flow for Zyzzyva can be seen in Figure 7.



(a) Gracious Execution



(b) Faulty Replica

Fig. 7. Zyzzyva's speculative execution. In panel (a), all replicas reply to the client with consistent results in a timely manner. In panel (b), replica 3 is faulty, so an extra phase of commit acknowledgement is required. [77]

Because Zyzzyva removes one of PBFT's phases (there is a pre-prepare phase and a second phase), the view change requires adjusting. To safely change views without three phases, an honest replica must not abandon a view until it knows every other correct replica will. To this end, another phase is added to the view change: a correct replica broadcasts a complaint when it suspects that the primary is Byzantine. Any replica that sees $f+1$ complaints knows to commit to a view change. This shifts costs from the agreement subprotocol to the view change subprotocol (so frequent view changes are performance-intensive). An additional adjustment is necessary to ensure that the replicas have a commit certificate when they are unanimous: replicas include all order requests (that is, the PRE-PREPAREs) since the last stable checkpoint in their view change messages, and a new honest primary extends the log with all requests that occurred in at least $f+1$ of the $2f+1$ view change messages they received. Note that the original protocol has a safety violation in the view change subprotocol that can be triggered by a faulty primary, which was only found 10 years later [78].

A related protocol, AZyzzyva, does not suffer from the safety violation in Zyzzyva [79]. It is essentially the same as Zyzzyva in the optimistic "common case," but it falls back on PBFT when detecting asynchrony or failures. That is, if a client does not receive $3f+1$ matching replies, it alerts the replicas, and the replicas then send signed message histories to the client. Upon receiving $2f+1$ of these, clients switch to the backup mechanism: send these unforgeable histories to the replicas, and the replicas use PBFT to order a pre-specified number of requests, including the requests from the signed histories. This requires more steps than Zyzzyva on the slow/recovery path and takes longer to switch back to the fast path, but it dramatically simplifies the codebase without a safety issue.

SACZyzzyva uses hardware-assisted trusted monotonic counters to improve upon Zyzzyva [80]. SACZyzzyva inherits the optimal resilience of Zyzzyva and eliminates the need for non-speculative fallback while only requiring a single replica – the primary – to use a trusted monotonic counter at any given time. There must be $f+1$ replicas with a trusted component to ensure that there is at least one correct replica that can be primary. The main idea is that the trusted monotonic counter value is attached to a message so that it is detectable if the sender equivocates due to the existence of a hole in the set of counter values. The primary uses the counter to bind a sequence of consecutive counter values to incoming requests to order them without communication between replicas.

4.4. A Permissioned DAG: Blockmania

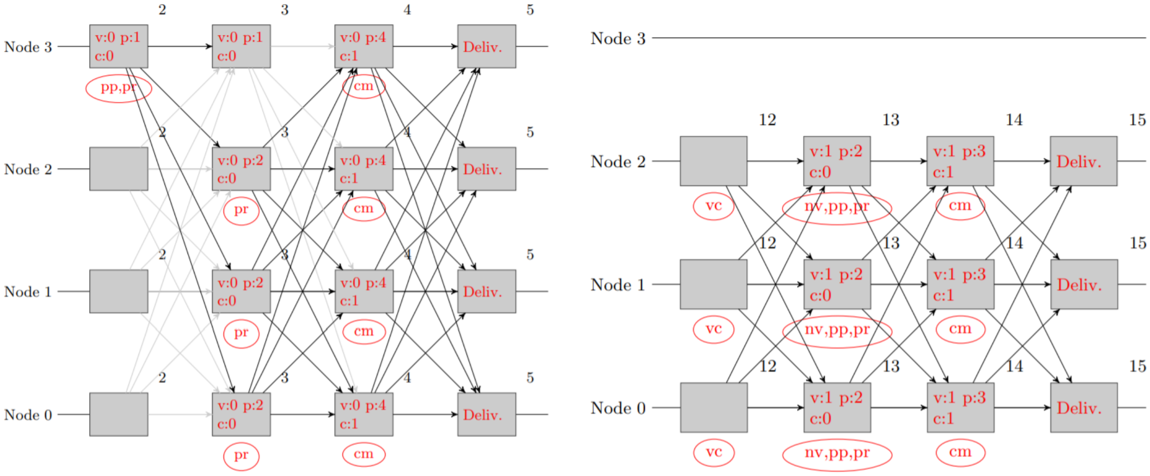
Blockmania is a partially synchronous BFT algorithm that effectively embeds PBFT-like messages inside of a DAG [81]. Essentially, there is a leaderless version of a PBFT state machine embedded inside the block content in the DAG, and interpreting the DAG allows for recreating a PBFT execution transcript without requiring the transmission of additional messages. This is possible because replicas "gossip about gossip," or tell each other everything they learn from every other replica (a similar approach is used in Hashgraph, which is discussed in Section 6.2). Every honest replica produces a (single) block in each round (hence, being leaderless). To form the DAG, honest replicas include references to all valid blocks they have seen, including contradictory ones, when they create their own blocks. Compared to PBFT, this approach reduces the worst-case communication complexity from $O(n^4)$ to $O(n^2)$. This performance improvement is because COMMIT, VIEW-CHANGE, and NEW-VIEW messages normally need evidence sent with them, but correctly interpreting the DAG recreates this evidence implicitly.

In Blockmania, blocks are referenced as (p,k), where p is the creator of the block, and k is its sequence number. The block's contents can include both a list of transactions and a list of references to all valid blocks received from other parties. Each replica also stores a current view number, $view_p$, and a list of input and output messages, in_p and out_p. Assuming no timeouts, the normal case operation of the protocol decides on position (p,k) by using a protocol similar to PBFT:

1. To propose a block B as a value for position (p,k), replica p broadcasts a PRE-PREPARE(p,k,B,v) message, where v is the view number, which begins at 0.
2. When a replica receives the first PRE-PREPARE message for view v, if the recipient's $view_p = v$, it broadcasts a PREPARE(p,k,B,v) message and adds the PREPARE and PRE-PREPARE messages to in_p. This is the only block that replica p will prepare in this view.
3. Replicas listen for PREPARE(p,k,B,v) messages and add them to in_p when $view_p \geq v$. Once the replica has $2f+1$ PREPARE(p,k,B,v) messages and the associated PRE-PREPARE in in_p, it broadcasts COMMIT(p,k,B,v).
4. Replicas listen for COMMIT(p,k,B,v) messages, adding them to in_p when $view_p \geq v$. Once the replica has $2f+1$ COMMIT(p,k,B,v) messages in in_p, the replica considers B to be decided at position (p,k). Note that B may be empty, or $\perp$.

However, timeouts can occur. This triggers a view change, which happens as follows:

1. The replica increases $view_p$ by one and broadcasts a VIEW-CHANGE$(p,k,view_p,S)$ message, where S is the set of all PREPARE and PRE-PREPARE messages supporting a block B that replica p is prepared on for position (p,k). If the replica has not locked on a value at that position, $S = \emptyset$. At this point, replica p stops participating in prior views except for potentially receiving more COMMIT messages.
2. Replicas wait for VIEW-CHANGE$(p,k,view_p,S)$ messages and add them to in_p if $v > view_p$. When $2f+1$ of these messages have been seen, the replica updates $view_p$ to v and broadcasts NEW-VIEW$(p,k,view_p,V)$, where V is the set of $2f+1$ VIEW-CHANGE messages.
3. When a replica sees the first NEW-VIEW(p,k,v,V) message where $v \geq view_p$ and $v > 0$, the replica sets $view_p = v$. The replica then checks if any VIEW-CHANGE messages included in V commit to a block B. If this is the case, the message is interpreted as a PRE-PREPARE(p,k,B,v) message; if not, the replica interprets it as a PRE-PREPARE$(p,k,\perp,v)$ message.
4. Replicas respond to the implied PRE-PREPARE accordingly and continue on with the protocol as normal.



(a) Correct execution **(b)** View Change

Fig. 8. Blockmania state machine interpretation example per block for position $(3,2)$. Each state machine includes a view number (v) and a count for prepare (p) and commit (c) messages received by the block in red. The *out* buffer is in the red circle below the blocks ('*pp*' for PRE-PREPARE, '*pr*' for PREPARE, '*cm*' for COMMIT, '*vc*' for VIEW-CHANGE, and '*nv*' for NEW-VIEW). [81]

Throughout this process, all sent messages are implicitly included in out_p. The protocol does not actually get executed via sending those messages directly. Instead, the protocol is inferred from the block graph, where each block is interpreted as including a set of PBFT state machines for positions that have not yet been decided. Denote a block at position (p,k) as $B_{(p,k)}$. Blocks are associated with the union of messages in the *out* sets of all state machines contained in the block. When a block $B_{(p,k)}$ includes a reference to another block $B_{(p',k')}$, a replica interprets this as replica p' sending replica p the messages that are included in the the *out* buffer of $B_{(p',k')}$. Those messages are then used to make progress in the PBFT state machines of block $B_{(p,k)}$ based on the ordered sequence of block references. New messages are added to the block's *out* buffer as the various state machines are interpreted while validating the block. When first attempting to decide on position (p',k'), the replica inserts a PRE-PREPARE$(p',k',B_{(p',k')},v=0)$ message, which is then included in its *out* buffer. Eventually, when a PBFT state machine embedded in the DAG decides, then the replica interpreting the state machine considers it decided as well. Once decisions are made for all n blocks for round k, a total ordering of transactions can be derived in some agreed upon way, such as the included fee.

See Figure 8 for an example of interpreting the DAG. Note that only the blocks are broadcast; the material in red is only interpreted from the block but never sent as a separate network communication. Figure 8a is a good execution, and Figure 8b shows a view change when replica 3 is faulty.

An advantage of interpreting a PBFT state machine rather than fully executing PBFT is that replicas can simply propagate blocks via gossip instead of having a complicated networking stack that handles the various message types. This can continue seamlessly even when view changes are occurring. In addition to the quadratic worst-case performance improvement over PBFT, Blockmania has low overhead due to interpreting blocks as messages themselves. Note that any deterministic BFT algorithm, not just PBFT, can be embedded in a communication DAG in this way [82].

5. Modern High-Performance Blockchains

Recently, a line of work has dramatically improved the simplicity and performance of BFT algorithms via pipelining and eliminating the need for a separate view change algorithm, among other innovations.

5.1. Streamlined Blockchains

The so-called "streamlined" blockchain is a generalization of a class of newer SMR consensus algorithms that are simple and highly performant [83, 84]. The simplicity of this approach makes it ideal for introducing more specific algorithms that have similar benefits, such as HotStuff, Casper-FFG, Tendermint, PiLi, and PaLa.

Classical BFT protocols had what might be called a "normal mode" and a "recovery mode." The normal mode is potentially simple: a designated proposer proposes a block by signing it, other replicas vote on it by signing it, and it becomes finalized upon receiving $\frac{2n}{3}$ distinct votes. This assures consistency when fewer than $\frac{n}{3}$ nodes are Byzantine, even if the proposer is Byzantine, regardless of network assumptions. However, a Byzantine proposer can stall liveness by not proposing anything or sending conflicting proposals, which necessitates a "recovery mode" (view change) to address liveness. Unfortunately, changes to the normal mode must be made to make the recovery mode work.

This motivates modern protocols that dispense with the view change entirely. Streamlined blockchains, such as the Streamlet protocol described below, provide an opportunity to elect a new leader in every epoch, so view changes are baked into the normal mode [84]. They support multiple proposer-election policies, including the more equitable policy of rotating every block and the more stable and performant policy in which rotation only occurs upon suspicion of misbehavior. Optimizations can be applied to the streamlined approach to improve upon the constant performance parameters presented here, which are not tight. Rather, the presentation is meant to be simple and easy to understand.

A partially synchronous version of the Streamlet protocol is presented here. Note that there is an implicit assumption that whenever a replica observes a new message, it echoes it to everyone else. In the following, a block is considered *notarized* upon receiving more than $\frac{2n}{3}$ votes from distinct replicas, and a chain is considered notarized when the blocks it includes are all notarized. The protocol follows a *Propose-Vote-Finalize* paradigm:

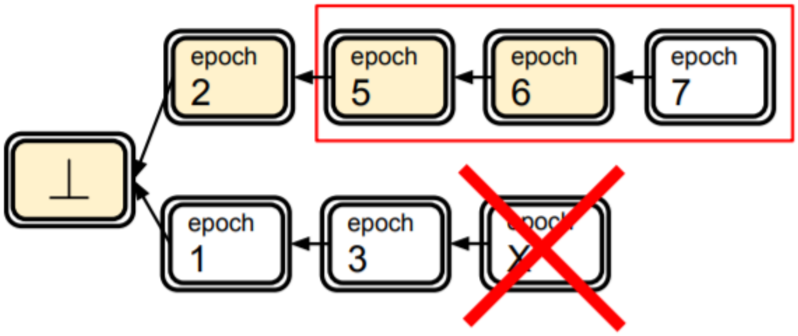


Fig. 9. Streamlet Finalization Rule. All blocks are notarized. The chain prefix up to the epoch-6 block is finalized ("$\perp -2-5-6$") once a replica has seen blocks 5, 6, and 7 built atop one another. There cannot be another block notarized at the same height as epoch-6, so chain "$\perp -1-3$" cannot grow further. [84]

1. **Propose**: The primary replica for the current epoch proposes a new block that extends from the longest notarized chain it has seen and breaks ties arbitrarily.
2. **Vote**: Replicas vote for the first proposal they see from the epoch's leader if the proposed block is appended to one of the longest notarized chains that the replica has seen. They do so by signing the proposed block.
3. **Finalize**: When there are three adjacent blocks with consecutive epoch numbers in any notarized chain, replicas finalize the prefix of the chain up to the second of the three blocks (see Figure 9).

To make the Streamlet protocol synchronous (and thus honest majority), this paradigm is tweaked by 1) decreasing the notarization requirement from $> \frac{2n}{3}$ to $> \frac{n}{2}$, 2) adjusting the finalization rule to require six consecutive epochs in a notarized chain in order to finalize the prefix of the chain with the last five blocks removed, and 3) requiring that the six blocks in consecutive epochs do not have conflicting notarizations at the same lengths at the time of finalization.

5.2. PiLi and PaLa

The protocols in this section follow the same *Propose-Vote-Finalize* paradigm with some adjustments in order to attain *optimistic responsiveness*. *Responsiveness* is the property that transactions are processed without reliance on synchrony. That is, processing depends on the actual network delay δ, which may be significantly less than worst-case delay: $\delta << \Delta$. Synchronous protocols usually have slow confirmation latency because Δ must be set conservatively for safety reasons. The idea of optimistic responsiveness – in which a protocol is responsive under certain good conditions – was introduced in [85], which showed that a $\frac{3n}{4}$ supermajority of honest and online replicas are necessary to attain this property for any honest majority protocol.

The PiLi protocol is secure against rushing adversaries in a weakly synchronous network model, which provides resilience to node churn [86]. It attains optimistic responsiveness when $\frac{3n}{4}$ honest nodes remain online and the proposer is among them. If $\frac{n}{2}$ honest nodes are online for the length of the confirmation delay, transactions are confirmed in an expected constant number of synchronous rounds. Consistency is guaranteed for all honest nodes, even those that drop offline sometimes, as long as the total number of offline and corrupt nodes are less than $\frac{n}{2}$. This makes the protocol more robust than classical synchronous consensus. Compared to the synchronous Streamlet protocol, PiLi gains these advantages at the expense of worse finalization latency.

In the PiLi protocol, a block is considered notarized upon receiving votes from the majority of voters. The finalization rule is that if a notarized chain ends with 13 consecutive epochs, the trailing eight blocks and the chain prefix are considered final. Define a *normal block* to be one where its epoch number is one greater than its parent's epoch number and a *skip block* to be one with an epoch number that is a multiple of 16 and at least 16 epochs apart from its parent.

In each epoch, the leader proposes a block that extends the *freshest* notarized chain it has seen (the freshest chain is the one whose chain tip has the higher epoch number). Replicas then vote on the first valid proposal they see, given that 1) the block extends from a parent block of an epoch no older than the freshest notarized chain that the replica had seen by the beginning of the previous epoch, and 2) if the block is not a skip block itself, then the replica has not observed any notarizations for conflicting blocks with the same epoch number for every block in the chain since the last skip block in the chain that the proposed block extends from.

Further, at the beginning of each epoch, replicas set a timer. If the timer goes off, replicas send each other timeout messages and advance to the next epoch upon receiving timeout messages from the majority of replicas. Alternatively, when an $epoch_e$ block receives votes from a $\frac{3n}{4}$ supermajority of voters, an $epoch_{e+1}$ block can be proposed and immediately voted on by the the leader of epoch $e+1$, thus achieving optimistic responsiveness.

For weak synchrony, replicas must only believe in "no-conflict" (that is, condition 2 above) if the majority of nodes confirm that belief rather than just using their own view. Voting on a block acts as an attestation that the replica has not seen recent conflicts, so a notarization on a block means that many replicas have not seen conflicts. Skip blocks prevent a denial-of-service attack where corrupt replicas double-vote, preventing honest nodes from voting due to conflict. Refusing to vote acts as a "complaint" that temporarily halts chain growth, but progress continues when an honest node produces a skip block during a period of synchrony.

A related protocol is the partially synchronous PaLa protocol, which tolerates $\frac{n}{3}$ corruptions [87]. For PaLa, notarization requires $\frac{2n}{3}$ votes. If two blocks are built on top of each other in back-to-back epochs, the first of them is finalized. The protocol description defines the terms "second" and "minute" using the constant $c = 6$, where a second is $c\Delta$ and a minute is $c^2\Delta$.

In each epoch e, a node is elected to be the proposer. If the proposer's chain ends with a block from epoch $e-1$, they immediately propose a new block. If their chain ends with a block from an epoch earlier than $e-1$ (that is, the prior proposer's block was not included), they wait for one second to potentially receive a fresher chain before proposing a new block. Whenever a replica during epoch e receives an $epoch_e$ block, it votes for the block if 1) it is consistent with the existing chain, 2) at least as fresh as the chain they saw at the beginning of epoch e, and 3) it has not previously signed a block for this epoch.

A replica advances to epoch e when it is currently in an epoch earlier than e and either of the following occur: it sees a notarized chain for epoch $e-1$, or it receives signed $timeout(e)$ messages from more than $\frac{2n}{3}$ replicas. Replicas set a timer for one minute upon entering epoch $e-1$ before broadcasting a $timeout(e)$ message.

Because the replicas must wait to collect notarizations for each block of the entire chain before voting on the next block, performance can be hindered significantly in high latency environments, even when there is considerable bandwidth available. This motivates the "doubly pipelined" variant of PaLa, which rotates leaders less frequently and has better performance. Here, replicas vote even if not all notarizations have been collected for the ancestor chain so long as the number of blocks at the tip that are not notarized is bounded by a security parameter k (with the above scheme having $k = 1$), and finalization requires k consecutively notarized blocks. For the current epoch, a proposer can repeatedly propose blocks so long as there are fewer than k proposed but not notarized blocks at the end of the chain.

PaLa allows committee reconfiguration to be done for half of the committee at a time, splitting the reconfiguration among two consecutive sets of k blocks to maintain safety when half of the committee has switched, assuming that each committee-half has fewer than $\frac{n}{3}$ corruptions.

5.3. HotStuff

HotStuff is a state-of-the-art high-performance BFT algorithm that utilizes pipelining while remaining secure in partially synchronous networks [88]. A synchronous version will be introduced later in this section [89], and an asynchronous version called VABA has also been proposed [90]. A variant of HotStuff is slated to be used in the Libra cryptocurrency (later renamed as Diem) [91].

The protocol has optimistic responsiveness, $O(n)$ communication complexity when the proposer is honest, $O(n)$ communication complexity for view changes, and $O(fn)$ worst-case complexity with f leader failures. That is, compared to PBFT, the view change procedure has a quadratic reduction in communication complexity. The cost of having a new leader drive consensus is no greater than that for the current leader, which supports frequent leader changes. These improvements come at the cost of adding an extra phase to each view. If the new pre-commit phase were removed, then liveness could be permanently stalled by an attacker. Note that there are similar protocols, such as Tendermint and Casper FFG, that make a different design choice and sacrifice optimistic responsiveness in order to avoid adding the extra phase.

The protocol defines a "quorum certificate" (QC) – sometimes also called a "commit certificate" – as $n-f$ votes on a proposal. These quorum certificates serve as proof that a sufficient number of replicas agree on something and may have different names depending on what they are used for.

The PBFT view change involves a proposer broadcasting proof from $2f+1$ replicas about their highest commit certificates, each of which includes $2f+1$ signatures. HotStuff improves upon this by a linear factor by including only one commit certificate and adding a rule that replicas only unlock a commit vote if they receive a commit certificate from a higher phase. An additional linear improvement to communication complexity comes from using aggregated threshold signatures instead of individual signatures and sending them to the leader instead of broadcasting them to everyone.

When entering a new view or after a timeout occurs, replicas transmit NEW-VIEW messages that include *prepareQC*, which is the highest QC for which a replica has voted to pre-commit. Replicas also store *lockedQC*, which is the highest QC for which the replica has voted to commit, and use the $SafeNode()$ predicate to decide whether a proposal is safe to accept. The predicate is true if either of the following holds: the proposal extends from the currently locked node (to maintain safety), or the proposal has a higher view number than the current *lockedQC* (to maintain liveness). The basic partially synchronous HotStuff algorithm is described here before pipelining is added:

1. **Prepare**: As leader, wait for $n-f$ NEW-VIEW messages, select the highest *prepareQC* included, and denote it *highQC*. The leader creates a proposal that extends from this block and broadcasts a PREPARE message with the proposal and *highQC* justifying its safety. Upon receiving a PREPARE message, other replicas decide to accept it or not using $SafeNode()$ and send a PREPARE message with a (partial) signature on the proposal to the leader.

2. **Pre-commit**: When the leader receives $n-f$ PREPARE votes on their proposal, they aggregate them into a *prepareQC* that is then broadcast in a PRE-COMMIT message. Replicas respond to the leader with a signed PRE-COMMIT message on the proposal.

3. **Commit**: When the leader receives $n-f$ PRE-COMMIT votes on their proposal, they aggregate them into a *precommitQC* that is then broadcast in a COMMIT message, and replicas respond with their COMMIT vote. At this point, replicas set their *lockedQC* to the *precommitQC* they received.

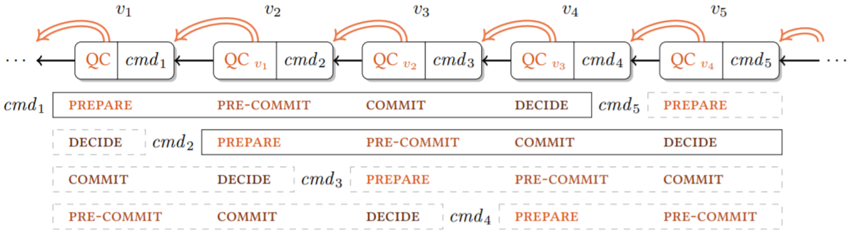


Fig. 10. Pipelining in Chained HotStuff. A quorum certificate is simultaneously used in different phases of the protocol to improve efficiency. [88]

4. **Decide**: When the leader receives $n - f$ COMMIT votes, they combine them into a *commitQC* and broadcast it in a DECIDE message. Upon receiving a DECIDE message, a replica considers the proposal linked to the *commitQC* to be committed, executes the commands, and moves to the next view.

The pipelined version of HotStuff is dubbed "Chained HotStuff" and works by having one epoch's commit step be executed during the next epoch's prepare phase. The idea is depicted in Figure 10.

Chained HotStuff improves upon the basic algorithm above by initiating a new view on every prepare phase. This not only allows pipelining but also reduces the scheme to two message types: NEW-VIEW and GENERIC. During the prepare phase of view v, the leader collects other replicas' votes into a *genericQC*, which is then sent to the next views' leader as part of delegating the next phase of the protocol. Instead of following through with a pre-commit phase, the new leader initiates a new prepare phase with its own proposal for view $v+1$. That is, this second prepare phase is the prepare phase for view $v+1$ while simultaneously satisfying the pre-commit phase for view v. This continues such that the prepare phase in view $v+2$ acts as the pre-commit phase for view $v+1$ as well as the commit phase for view v.

Each block contains a quorum certificate which may or may not point to the direct parent block of the chain. When a block $b*$ has a QC that refers to (and thus *justifies*) a direct parent, it is called a "One-Chain." That is, $b*.QC.block = b*.parent$. Define $b'' = b*.QC.block$. Then block $b*$ is a Two-Chain if it is a One-Chain and $b''.QC.block = b''.parent$. It forms a Three-Chain if b'' forms a Two-Chain. There may also be gaps, which represent leader failures. Assume the chain of justifications is $b \leftarrow b' \leftarrow b'' \leftarrow b*$. When $b*$ is a One-Chain, the prepare phase of b'' has succeeded. When $b*$ is a Two-Chain, then the pre-commit of block b' has succeeded. When $b*$ is a Three-Chain, the commit phase of block b has succeeded, and b is committed and final. This is illustrated in Figure 11.

In more detail, Chained HotStuff works as follows. For each view v, the leader of the view will update their *highQC* based on NEW-VIEW messages received and update *genericQC* if *highQC* is higher. They then create a proposal $b*$ and begin the prepare phase by sending a GENERIC message.

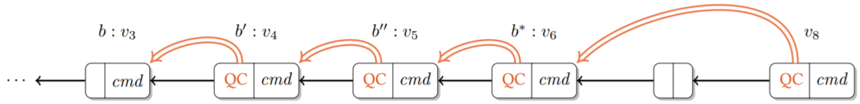


Fig. 11. Chained HotStuff justification. The blocks for v_4, v_5, and v_6 form a Three-Chain, whereas v_8 fails to form a One-Chain. v_8 also shows a quorum certificate that justifies a block that is not the direct parent of the block proposed in that view. Instead, the parent is a dummy block that represents the leader failure in v_7. [88]

Each replica then checks the predicate $SafeNode(b*)$, and if it evaluates to true, they send a GENERIC vote on $b*$ to the leader of view $v+1$. If $b*.parent = b''$, the replica updates $genericQC$ to $b*.QC$. Here, $b*$ is a One-Chain, so the prepare phase for b'' has succeeded, and the replica begins to pre-commit on b''. If $b*.parent = b''$ and $b''.parent = b'$, the replica updates $lockedQC$ to $b''.QC$. This means that $b*$ is a Two-Chain, so the replica enters the commit phase of $b*$'s grandparent, b'. If $b*.parent = b''$, $b''.parent = b'$, and $b'.parent = b$, then the replica executes commands through b and responds to clients. That is, when $b*$ is a Three-Chain, the replica enters the decide phase on $b*$'s great-grandparent, b. The leader of the next view waits for all messages until they see $n-f$ votes on a proposal and then updates their own $genericQC$.

5.3.1. Sync HotStuff

There is also a HotStuff variant intended for synchronous and weakly synchronous networks [89]. This variant has nearly optimal latency of 2Δ ($+O(\delta)$), optimistic responsiveness with $> \frac{3n}{4}$ honest replicas (which requires larger QCs than those used above), and tolerates up to $\frac{n}{2}$ corruptions. Sync HotStuff removes the need for lock-step execution that PiLi has and thus dramatically reduces finalization latency from what would otherwise be between 40Δ and 65Δ. The key trick is to move the synchronous waiting periods off the critical path, and the only step that requires waiting $O(\Delta)$ time is to check for leader equivocation before committing.

Leaders propose blocks, and then replicas vote on the proposal as soon as they have seen it so long as they have not seen an equivocating proposal. Replicas then start a timer for 2Δ. The proposal is committed if the timer goes off, the view has not changed, and there has not been equivocation. Replicas begin timers for subsequent heights without waiting for the previous height to commit, so they may have numerous commit timers running simultaneously at different heights. View changes are initiated upon detecting equivocation or the leader failing to propose a block within 2Δ.

For weakly synchronous networks, the above is changed so that the timer to commit block k begins after receiving $f+1$ proposals for block $k+1$ (which has the certificate for k). When timeouts occur, replicas broadcast commit messages for block k, and replicas commit after seeing $f+1$ commit messages.

In Sync HotStuff, if a replica votes at time t, the commit is considered safe at time $t+2\Delta$. Waiting for 2Δ provides safety because the replica's vote reaches all honest replicas by $t+\Delta$, at which point they will not vote for an equivocating block. If another replica voted for an equivocating block before $t+\Delta$, the first replica would have seen it by $t+2\Delta$. This also implies that all honest replicas will have voted by $t+\Delta$, so a QC will form by $t+2\Delta$, ensuring safety.

An earlier version of the protocol suffered from a "force-locking" attack that removed safety from the synchronous version and liveness from all versions but has since been fixed [92].

5.4. Further Optimizing Latency

It is possible to obtain optimistic responsiveness with lower latency than the protocols described above using protocols similar to PiLi and Sync HotStuff [93]. The challenge of optimizing for latency is in handling the situation where it is unclear whether the fast-path optimistic conditions are met. If it is clear that the optimistic conditions hold, there are latency-optimized protocols to use for that setting.

When an optimistically responsive commit rule and a slower synchronous commit rule exist in synchronous protocols, the sum of the latencies of the two rules must be at least 2Δ, such that when a faster optimistic commit rule is used, the synchronous commit rule must be correspondingly slower to make up for it [93]. For example, if the optimistic commit path requires 0.5Δ to commit, then the synchronous path requires at least 1.5Δ to commit. In Sync HotStuff, the synchronous slow-path latency is 2Δ, while the fast path commit latency is 2δ, so it is not fully optimal. Further, switching between the optimistic and the slow paths imposes its own delays of at least Δ. The protocols described in [93] achieve optimistic responsiveness and optimal latency simultaneously by removing the explicit switch between fast and slow paths and making it such that replicas need not know which path they are on; both progress simultaneously. Alternatively, the Apollo SMR protocol has an optimistic finalization latency of $(f+1)\delta$ and commits a block every δ time units [94] while maintaining linear communication complexity in the best case. It achieves this by having a block committed whenever $f+1$ blocks are built on top of it, which obviates the need to detect equivocation.

6. Asynchronous BFT

The protocols described up to this point in the document all fail to maintain liveness when messages may be arbitrarily delayed and synchronous ones further lose their safety. Even when partially synchronous timing assumptions hold in practice, performance degrades rapidly in an unpredictable network, and throughput fails to quickly recover after a temporary network partition. In contrast, asynchronous protocols are able to continue confirming transactions at network speed as protocol messages arrive, regardless of the message delay or ordering. These protocols do not rely on timeout mechanisms, which may make them easier to implement.

To see how liveness fails under partial synchrony, one may consider the following adversarial network scheduler attacking an execution of PBFT, which results in zero throughput. Assume that a single replica has suffered a crash fault. The adversarial scheduler induces message delays whenever the leader is honest, causing an eventual view change on timeouts and thus moving to the next leader. At some point, the crashed replica becomes the next leader, and the scheduler quickly delivers all messages between honest replicas. Unfortunately, because the leader is crashed, there is still no progress made. The existence of synchronous periods where messages are delivered is insufficient to maintain liveness because they occur at times when the algorithm is unable to make use of them. An asynchronous protocol, on the other hand, would make progress and confirm transactions during these synchronous periods.

Because the timeout interval (usually) increases exponentially in partially synchronous protocols, even a limited version of this network attack would dramatically slow the network's recovery from a partition. If a replica is crashed, the network is partitioned for a length of $2^D\Delta$, and the scheduler heals the network when the crashed node is supposed to become leader, then there is a $2^{D+1}\Delta$ delay before starting a new view change despite the network being synchronous. Again, an asynchronous protocol would begin confirming transactions immediately during this period.

6.1. HoneyBadgerBFT

The first practical asynchronous BFT algorithm designed was the leaderless HoneyBadgerBFT, which utilizes randomization to overcome FLP impossibility [95]. HoneyBadgerBFT provides security against static adversaries but can be secure against adaptive adversaries by substituting in different cryptographic primitives at the cost of worse performance [96]. The primary novelty is a reduction from atomic broadcast to the asynchronous common subset (ACS) primitive, which was improved from $O(n^2)$ to $O(1)$ (the prior state-of-the-art was from [97]) but also uses threshold decryption for censorship resistance and improving liveness.

An ACS protocol allows each replica to propose a value and then ensures that every replica outputs a common vector that includes the input values from at least $n-2f$ correct replicas. One can create a state machine replication protocol by simply having each replica propose a set of transactions as input into the ACS primitive with the output being the union of the transactions. ACS has the following requirements:

- Agreement – If an honest replica outputs v, then every replica outputs v.
- Validity – If an honest replica outputs a set v, then $|v| \geq n-f$ and v contains the inputs of at least $n-2f$ honest replicas.

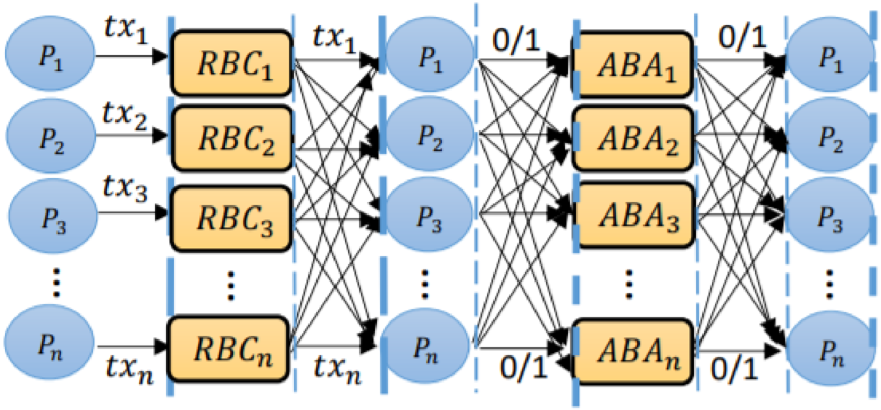


Fig. 12. HoneyBadgerBFT's ACS structure. There are n instances of reliable broadcast followed by n instances of asynchronous binary agreement. [98]

- Totality – If $n-f$ honest replicas receive an input, then all honest replicas produce an output.

ACS protocols typically involve two phases: 1) reliable broadcast to distribute each replica's proposed values and 2) asynchronous binary Byzantine agreement (sometimes abbreviated as ABA or ABBA) to decide on a bit vector that indicates which reliable broadcasts have completed. The structure of HoneyBadgerBFT's ACS protocol can be seen in Figure 12.

The HoneyBadgerBFT algorithm is composed of a variety of subprotocols, including an expected constant-time ABA protocol, threshold signatures to produce the common coins required for the ABA protocol, a bandwidth-efficient reliable broadcast protocol, and threshold decryption. The protocol idea is simple: if the maximum block size is B and there are n replicas, each replica will randomly choose $\frac{B}{n}$ transactions from their mempools and encrypt them. This encrypted set is then passed into the ACS protocol, which outputs the common subset of transactions. Each replica then performs its share of decryption on the common subset and broadcasts its decryption share. Upon receiving $f+1$ decryption shares, replicas fully decrypt the transaction set. The block that results is the canonically sorted set of transactions.

Note how efficiency is improved by having honest replicas try to propose disjoint subsets. By having replicas pick transactions randomly, it is expected that each transaction is only proposed by a single replica. The addition of threshold decryption prevents some types of transaction censorship because replicas will not know which transactions were proposed by whom. A trade-off to this is that a faulty replica may propose invalid transactions in the ACS, which would not be detected until after they are included in a block. This means that after transactions are ordered, the invalid transactions must be removed from the block or at least remain unexecuted.

For the reliable broadcast subprotocol, HoneyBadgerBFT actually employs a more sophisticated primitive called asynchronous verifiable information dispersal (AVID). This primitive was first presented in [99] and combines asynchronous reliable broadcast and erasure coding. The use of erasure coding significantly reduces communication complexity for large messages compared to Bracha's broadcast protocol described in Section 1.2.

6.1.1. Mostéfaoui et al.'s Asynchronous Binary Agreement Protocol

Mostéfaoui et al. introduced a core component of HoneyBadgerBFT and some related protocols: asynchronous binary agreement (ABA), where agreement occurs over a single bit [100]. Note that the "broadcast" protocols described here do not meet the standard definition used in Section 1.2 because each party transmits a value.

The protocol is built in layers with the most basic component being a binary value (BV) broadcast protocol, which eliminates from consideration any value that was broadcast only by malicious parties. Each correct process p_i BV-broadcasts a binary value and eventually obtains a set of binary values stored in bin_values_i. Let $witness(v)$ be the number of different processes from which $B\_VAL(v)$ was received, and let $bin\_values_i = \emptyset$. The operation $BV\_BroadcastMSG(v_i)$ consists of multicasting $B\_VAL(v_i)$ and then returning. Then, when p_i receives $B\_VAL(v)$, it does the following:

1. If p_i has not yet multicast $B\_VAL(v)$ to other replicas, and if $witness(v) \geq f+1$, then p_i multicasts $B\_VAL(v)$. Note that a process echoes a value like this only once.
2. If $witness(v) \geq 2f+1$ and $v \notin bin\_values_i$, then p_i locally delivers the value by setting $bin\_values_i \leftarrow bin\_values_i \cup \{v\}$.

The next layer is strong binary value (SBV) broadcast, which synchronizes processes such that if a single value v is delivered to an honest process, then v is delivered to all honest processes. The operation $SBV\_BroadcastMSG(v_i)$ begins by invoking $BV\_BroadcastMSG(v_i)$ and waiting until bin_values_i is non-empty.

When the waiting stops, bin_values_i may not be at its final value. Nevertheless, p_i then multicasts a message $AUX(w)$, where $w \in bin\_values_i$. If there is more than one value in bin_values_i, any value w will do. Then p_i waits until there exists a set $view_i$ such that its values belong to bin_values_i and come from $AUX()$ messages received from $n-f$ distinct processes. Finally, the algorithm returns $view_i$.

The third layer is a double-synchronized binary value (DSBV) broadcast algorithm. The goal of this algorithm is to replace the potentially distinct values v and w that are broadcast by honest replicas by at most one of v or w, plus potentially the default value denoted $\perp$. As with SBV, this returns a $view_i$, but it can now contain a default value $\perp$ and – at most – only one of the two possible binary values. DSBV broadcast reduces the power of Byzantine replicas such that no view delivered by an honest replica includes values broadcast only by Byzantine replicas (from BV broadcast) and that if a view delivered to an honest replica contains only the single value v, then v will be included in the views of all honest replicas (from SBV broadcast). The operation $DSBV\_BroadcastMSG(v_i)$ works as follows:

1. $view_i[0] \leftarrow SBV\_BroadcastSTAGE[0](v_i)$.
2. If $view_i[0] = \{v\}$, then $aux_i \leftarrow v$. Else, $aux_i \leftarrow \perp$.
3. $view_i[1] \leftarrow SBV\_BroadcastSTAGE[1](aux_i)$.
4. Return $view_i[1]$.

That is, DSBV broadcast consists of two stages of SBV broadcast. When the first SBV broadcast returns, it is possible that each of the two binary values are represented in $view_i$. This occurs when honest replicas input different values to their SBV broadcast operation. In line (2) of the algorithm, replicas filter the output from the first SBV broadcast to determine their input into the second instance. While two possible values – say, a and b – were possible inputs into the DSBV broadcast (and therefore the first SBV broadcast), the second SBV broadcast will only include one of those inputs (which must have been proposed by an honest replica) and $\perp$.

The final ABA algorithm utilizes this DSBV broadcast operation and a *weak common coin*. Common coin protocols result in some random output that can be observed by all participants and is unpredictable to an adversary. For a weak common coin protocol, there is a constant probability that the replicas involved will see different random values returned by the functionality. In the description of the ABA protocol that follows, the weak common coin is invoked by calling $CCRandom()$.

The ABA algorithm begins when a process p_i invokes $propose(v_i)$, where $v_i \in \{0,1\}$. The operation begins by setting est_i (p_i's current estimate of the decision) to v_i and setting the round number, r_i, to zero. Processes proceed in asynchronous rounds, where each round is two phases, and each phase is made up of a single DSBV broadcast instance. Let $view_i[r_i, 1..2]$ be the local results of the DSBV broadcast instance at round r_i. The ABA algorithm repeats the following sequence indefinitely:

- $r_i \leftarrow r_i + 1$.
- Phase 1: Help replicas agree on a single value.
    1. $view_i[r_i, 1] \leftarrow DSBV\_BroadcastPHASE[r_i, 1](est_i)$.
    2. $b_i[r_i] \leftarrow CCRandom()$.
    3. If $view_i[r_i, 1] = \{v\}$ and $v \neq \perp$, then set $est_i \leftarrow v$. Otherwise, $est_i \leftarrow b_i[r_i]$.
- Phase 2: Help replicas recognize when they have reached a round where all honest replicas will forever maintain the same estimate.
    1. $view_i[r_i, 2] \leftarrow DSBV\_BroadcastPHASE[r_i, 2](est_i)$.
    2. If $view_i[r_i, 2] = \{v\}$, then decide v if not yet done.

3. Else if $view_i[r_i, 2] = \{v, \perp\}$, then $est_i \leftarrow v$.
4. Else if $view_i[r_i, 2] = \{\perp\}$, then skip to the next round.

As presented, the ABA algorithm is non-terminating. To make it guarantee termination, whenever a process decides, it also terminates and multicasts a message $TERM[r](v)$ to tell other processes it is done. That is, replace "decide v if not yet done" with "multicast $TERM[r_i](v)$ and return v." It also requires mild tweaks to the SBV and BV broadcast algorithm to accommodate waiting for these messages.

6.1.2. Reducing HoneyBadgerBFT's latency with BEAT

Duan et al. present BEAT – a family of five asynchronous BFT protocols with different trade-offs, all of which outperform HoneyBadgerBFT in latency and some of which improve throughput [101]. These protocols follow a similar structure as HoneyBadgerBFT but use more efficient labeled threshold cryptography and erasure coding schemes to improve performance.

Of the five algorithms, BEAT0, BEAT1, and BEAT2 are most relevant to this document because they are for general state machine replication. In contrast, BEAT3 and BEAT4 are "BFT storage" algorithms that do not require replicas to maintain the full system state. BFT storage only allows read and write operations to a key-value store, and the state is erasure-coded, so it cannot support on-chain smart contracts.

The base algorithm, BEAT0, uses labeled threshold encryption to make transactions uniquely identifiable. This makes it easier for replicas to ignore duplicate transactions that had been input to the ACS subprotocol. Further, instead of HoneyBadgerBFT's use of threshold signatures to derive the common coin, BEAT0 uses a more efficient threshold pseudorandom function. Finally, it uses more efficient erasure-coding. The other BEAT algorithms use these improvements as well.

BEAT1 replaces the AVID reliable broadcast primitive used in HoneyBadgerBFT with Bracha's broadcast. This induces a trade-off because it increases asymptotic communication complexity. However, in situations where there is little contention and small batches (i.e., small blocks), Bracha's broadcast has significantly lower latency than AVID.

BEAT2 uses Bracha's broadcast as well. Additionally, it optimistically moves the threshold encryption to the client instead of the replicas. This substantially reduces latency because the threshold encryption is one of the biggest drivers of latency in HoneyBadgerBFT. The trade-off is that there is a weaker version of liveness in which servers may be able to censor specific clients. However, full liveness is restored by using an anonymous communication network like Tor.

The asymptotic communication complexity with n replicas ordering a set of transactions of size B is $O(nB)$ in HoneyBadgerBFT and BEAT0. BEAT1 and BEAT2 worsen this to

$O(n^2B)$. BEAT3 and BEAT4, which are not covered in this document, reduce this complexity significantly to $O(B)$ by having agreement occur on a constant-sized checksum instead of all data in the block as well as some other techniques.

6.1.3. Improving ACS Performance with Dumbo

HoneyBadgerBFT's asynchronous common subset protocol, as shown in Figure 12, requires n instances of reliable broadcast (RBC) to be run so that each replica can propose their own set of transactions, followed by n instances of asynchronous binary agreement to decide on each of those inputs. In more detail, whenever a replica delivers a value that was broadcast by its peer P_i, the replica starts the i-th ABA instance with an input of 1. When any honest replica receives 1 from $n-f$ ABA instances, it inputs 0 to the remaining instances and moves on.

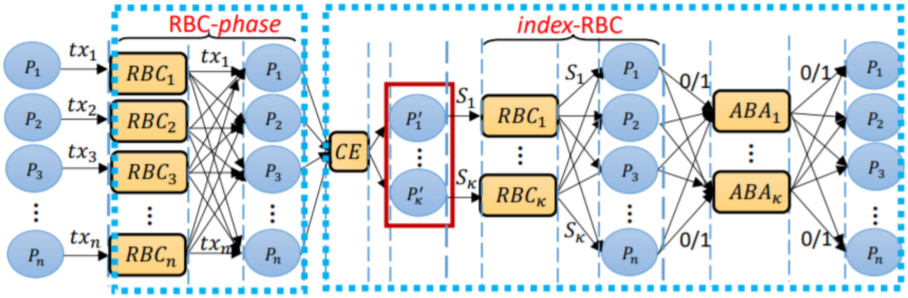
Because the ABA protocol is randomized, the expected number of rounds of each ABA instance is constant. However, running n instances at the same time can blow up the expected number of rounds considerably. This requirement of executing n ABA instances is a major performance bottleneck for the protocol, particularly because the slowest ABA instance determines the running time to finish the ACS execution. This problem is exacerbated by having different ABA instances start at different times, depending on when each instance of reliable broadcast delivers its value.

The Dumbo family of protocols improves upon HoneyBadgerBFT's performance by modifying the ACS protocol to require substantially fewer instances of the ABA subprotocol to be run [98]. Two particular improvements are suggested:

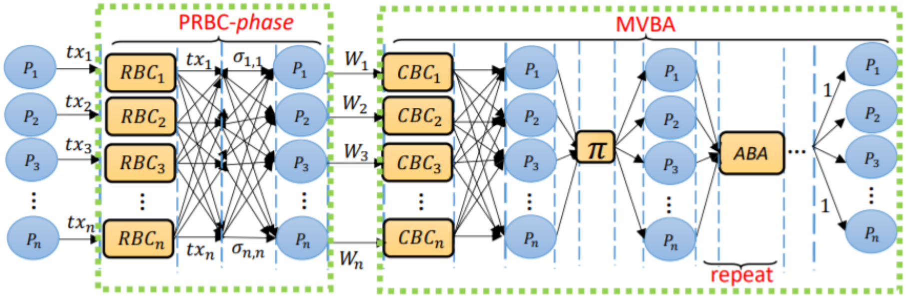
1. Rather than having an ABA instance per replica, a small set of κ "aggregators" can be chosen to lead the reliable broadcast instances, such that only κ instances of ABA must be run in the Dumbo1 protocol.
2. Careful use of multi-valued validated Byzantine agreement (MVBA) can further reduce the number of ABA instances required to a constant (an expected three consecutive rounds of ABA).

MVBA was rejected in HoneyBadgerBFT because it has high communication complexity when run over values that may be of large size, such as a block of transactions. However, a proper strategy would allow MVBA to be run on smaller values, as is done in Dumbo2. Part of the trick involves a new primitive – provable reliable broadcast (PRBC) – that outputs a short proof that at least one honest peer has received the input to the reliable broadcast. The first improvement can be seen in Figure 13a, and the second is displayed in Figure 13b. Note that the insights from BEAT0-BEAT2 can be applied to improve Dumbo as well.

Dumbo1 requires two phases of reliable broadcast. The first phase is the same as HoneyBadgerBFT, where replicas broadcast their input sets of transactions. At this point, if there were an honest leader, they could wait until $n-f$ RBC instances have completed and then



(a) ACS of Dumbo1



(b) ACS of Dumbo2

Fig. 13. ACS structure of Dumbo protocols. RBC is reliable broadcast, ABA is asynchronous binary agreement, CE is committee election, PRBC is provable reliable broadcast, MVBA is multi-valued validated Byzantine agreement, CBC is consistent broadcast (a relaxation of reliable broadcast), and π is a permutation. [98]

simply tell the other replicas what the results were. By selecting κ leaders, the protocol can ensure that there is at least one honest leader with high probability. These κ leaders then begin a second round of RBC called index-RBC, where they broadcast the set of indices of the $n-f$ values that the leader has already received. That is, the inputs into the index-RBC instances are just small subsets of indices, denoted S_i, rather than the actual transactions. The final phase is to run κ ABA instances, where an honest replica will input 1 to the i-th instance if it has already seen S_i and all of the corresponding values from the first round of reliable broadcasts. At least one honest replica will receive all of the input values that correspond to S_i, so reliable broadcast ensures that all honest replicas will eventually deliver those values as well. When an honest replica outputs 1 for the i-th ABA instance, all honest replicas are eventually guaranteed to output 1 as well. The overhead for this procedure is κ additional RBC instances and a single coin tossing for committee election (CE in 13a), but the ABA savings dominate this overhead.

While Dumbo1 reduces the number of ABA instances needed, it is still necessary for κ instances to be executed. In theory, this is still $\kappa - 1$ "too many" instances. If it were possible to identify a single, correct input vector, this could be further reduced. This motivates the use of MVBA in Dumbo2. In MVBA, each replica submits an input value, and the output is a single value that satisfies a predefined predicate. Since MVBA is inefficient when there are large inputs, and the inputs to the ACS protocol may indeed be large, it is necessary to figure out how to safely reduce the size of the MVBA input. This is done through the clever use of a short "indicator" value, denoted W_i in Figure 13b, which is used as the MVBA input instead of the much larger original data. The MVBA protocol will output a single indicator that honest replicas use to work backward to select the corresponding reliable broadcast instance. Unfortunately, honest replicas may end up outputting the indicator from a Byzantine replica. This problem is addressed through the use of a new primitive, *provable* reliable broadcast (PRBC), which provides a succinct proof that at least one honest replica received the input. PRBC can be realized through the use of threshold signatures on the RBC index: when honest replicas receive a value from a sender, they multicast a signature share on the index and can construct the full signature as the proof upon seeing $f+1$ signature shares for the same index.

The indicator value will include a set of these proofs and their corresponding RBC indices. The predicate used for MVBA requires that $n-f$ of these indices exist and that their corresponding proofs are valid. As a result, an honest replica knows that the transactions corresponding to the included RBC indices were received by enough replicas to know that at least one (other) honest replica received them and that all honest replicas will eventually receive them. Therefore, if an honest replica initiated MVBA after seeing $n-f$ PRBC instances complete, all honest replicas simply need to wait to receive the values from the PRBC phase.

The Dumbo protocols can also be combined with an optimistic fast path to substantially improve latency when network conditions are good [102]. In this case, a much faster consensus protocol is executed, and Dumbo is used as a fallback mechanism in a way that is similar to a view change.

6.2. An Asynchronous Permissioned DAG: Hashgraph

The protocols above allow a set of replicas to maintain a sequentially ordered log of transactions or batches of transactions in the form of a blockchain. This section explores Hashgraph – an asynchronous BFT protocol that operates over a DAG. Like HoneyBadgerBFT, Hashgraph is a leaderless protocol that uses randomization to achieve consensus with probability 1 [103]. The protocol is similar to Blockmania (Section 4.4) in that it involves "gossip about gossip" and utilizes "virtual voting" to interpret the resulting communication graph (called a *hashgraph* throughout this section). That is, replicas tell each other about the communications that they have received from other replicas, and the communications themselves are used to infer how the replicas would vote.

In the Hashgraph algorithm, an *event* is a vertex in the hashgraph structure that is signed by its creator. When Bob gossips to Alice and sends her his current view of the hashgraph, an event is created and signed by Alice. This event contains the hashes of Alice's most recent event (called its *self-parent*), Bob's most recent event prior to the gossiping, and a timestamp. It can also include transactions that Alice proposes to the network. The algorithm occurs in rounds, and the first event that any given replica creates in a round is called a *witness*. Only witness events are considered in virtual voting. A *famous witness* is a witness that – according to the hashgraph – was received by a supermajority of replicas by the start of the next round. Byzantine agreement is sought over the witnesses rather than all events, and transactions are ordered based on whether witnesses are famous or not.

A few more terms must be defined before presenting the algorithm. An event x is an *ancestor* of event y when x is y or when x can be reached from y through any sequence of parent relationships. Further, x is a *self-ancestor* of y if x is y or when x can be reached from y by a sequence of self-parent relations. A pair of events – (x,y) – is a *fork* if x and y were created by the same replica, but neither is a self-ancestor of the other. Forks reflect inconsistencies in the hashgraph. An event x can *see* event y when y is an ancestor of x, and the ancestors of x do not include a fork by the creator of y. Finally, an event x *strongly sees* event y when x can see y, there exists a set of events by more than $\frac{2n}{3}$ replicas such that x can see every event in the set, and every event in the set can see y. Figure 14 shows an example of strongly seeing.

The security of Hashgraph can be argued based on quorum intersection. In particular, if a pair of events (x,y) is a fork, and x is strongly seen by event z in hashgraph A, then y will not be strongly seen by any event in any hashgraph B that is consistent with A. The Hashgraph protocol itself involves replicas running two loops in parallel:

1. Pick a random replica, gossip to them all known events, and create another event to record this gossip.
2. Receive gossip from another replica, create a new event, and then call three subpro-

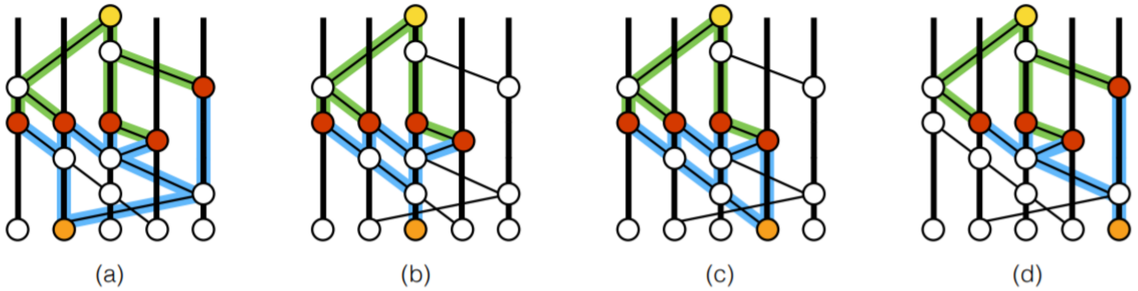


Fig. 14. Hashgraph strongly seeing. Time advances from the bottom of the hashgraphs to the top. The yellow event at the top of each hashgraph can strongly see an orange event on the bottom row. In each hashgraph, the orange event is an ancestor of four (or five) red events by different block creators, each of which is an ancestor of the yellow event. If the four orange events and both parents of the yellow event were created in round r, then the yellow event is created in round $r+1$ because it strongly sees more than $\frac{2n}{3}$ witnesses created by different replicas in round r. [103]

cedures: $divideRounds()$, $decideFame()$, and $findOrder()$.

- *divideRounds()*: For every event x, the replica sets r as the maximum round of the parents of x. Then, if x can strongly see more than $\frac{2n}{3}$ of the witnesses from round r, the replica sets x's round to be $r+1$. Otherwise, it is round r. Regardless, the replica then determines whether x is a witness by checking if x has no self-parent or if x's round is greater than the round of x's self-parent.

- *decideFame()*: This is the step where Byzantine agreement occurs. For each witness that has been identified, the replica checks whether it is a famous witness. Given a witness x in round r, each witness from round $r+1$ votes (implicitly) that x is famous if it can see it. If more than $\frac{2n}{3}$ agree that x is a famous witness, agreement has been found and the election is over. This continues for as many rounds as necessary to reach agreement. In normal rounds, the witness votes in line with the majority of the witnesses it strongly sees from the prior round. There are also periodic rounds where the vote is determined via a coin flip in order to get around FLP impossibility when networks are asynchronous.

- *findOrder()*: At this point, all witnesses from round r have had their fame decided. The replica removes any famous witness with the same creator as any other in that set. The unique famous witnesses that remain are used to totally order events. An event has a "received round" of r if that is the first round in which every unique famous witness is a descendent of the event, and the fame of each witness was decided by round r. The timestamp of the event is taken as the median of the timestamps of the events where each replica first received it. Events are then sorted by received round with ties broken by the median timestamp and further ties broken based on the signature on the event XORed with the signatures of the unique famous witnesses in the same received round.

One of the alleged advantages of the Hashgraph protocol is "fairness" in transaction ordering (see Section 7.1). In particular, it is claimed that it would be challenging for an attacker to "manipulate which of two transactions will be chosen to be first in the consensus order" [103]. Unfortunately, it fails to achieve this, as there is a method that allows an adversary to determine the order of transactions by exploiting the use of the median of the timestamps [104].

Other asynchronous DAG-based protocols that interpret a communication graph in this way exist, including Aleph [105] and DAG-Rider [106]. Aleph improves upon Hashgraph by using a more efficient binary agreement subprotocol than the *decideFame()* procedure above [105]. DAG-rider improves upon both Hashgraph and Aleph by removing the need for signatures to maintain safety and by providing *eventual fairness*, such that all block proposals by honest replicas are guaranteed to eventually be included in the ledger [106].

7. Miscellaneous Permissioned BFT

This section demonstrates some additional properties attainable by or techniques useful for BFT algorithms. These properties go beyond agreement and liveness to provide extra guarantees or functionality. For example, protocols can prevent replicas from manipulating the order of transactions, detect and expel replicas that behave maliciously, or dynamically reconfigure the set of replicas participating in consensus. Other protocols give certain replicas special roles to enhance performance, allow clients to have flexibility with respect to beliefs about network assumptions or the number of faults, or favor availability over consistency during network partitions (while maintaining deterministic finality). The details of the protocols here are omitted.

7.1. Fairly Ordering Transactions

There are situations in which the ability to choose the order of transactions included in the ledger provides an unwarranted or undesirable advantage to a replica. A major problem in traditional finance is *front-running*, where a participant uses the knowledge of somebody else's transactions for selfish advantage in their own dealings. For example, Bob may submit an order to his broker Alice to purchase 1000 shares of stock in company A. Before Alice fills the order, she can submit her own order to purchase 1000 shares and immediately turn around and sell the 1000 shares to Bob at a higher price for a guaranteed profit. For many distributed applications, the privileged position of block producers that choose the order of transactions makes them a potential adversary. Some consensus protocols attempt to reduce the ability of replicas to perform this and other kinds of manipulation by reducing the power of replicas to order transactions [104, 107–109].

An early attempt to achieve fairness in transaction ordering was the Helix protocol [108], which used threshold cryptography to limit the ability of replicas to censor transactions proposed by each other. Threshold decryption is used to create a randomness beacon that is then used to elect the next PBFT-esque committee in an unpredictable way and to force replicas to randomly sample the choice of transactions they include. Because clients encrypt transactions before sending them to replicas, the replicas lack information that would be useful to exploit ordering. However, this allows clients to arbitrarily spam the network with invalid transactions, so it may require identity verification for clients.

More recently, the Aequitas family of protocols [104] attempted to provide a property dubbed *order fairness*, such that if sufficiently many replicas receive transaction tx_1 before a different transaction tx_2, then all honest replicas will output tx_1 before tx_2. While it was proven that this property is impossible *within* blocks, the Aequitas protocols can provide this property *between* blocks. That is, if enough replicas receive tx_1 before tx_2, then tx_2 will appear in either the same block as tx_1 or a later block. The protocols rely on a primitive – FIFO-broadcast – that guarantees that broadcasts are delivered by honest replicas in the same order in which they were originally broadcast. Replicas use FIFO-broadcast to gossip their local transaction ordering and then later come to agreement on the set of replicas whose local orderings should be considered for a given transaction before ultimately finalizing the global ordering based on these local orderings.

A variety of similar fairness-related definitions can be found in [107], which presents a set of protocol extensions or "widgets" that provide fairness. They can be added as a preprocessing step to blockchain protocols at the expense of increased latency. As with Aequitas, true fairness is impossible because it would require blocks to be of potentially unlimited size (this is a trade-off that the Aequitas protocols accept). A variety of more relaxed notions are possible, including that fairness be provided with a fixed probability or that if all honest parties saw a transaction tx_1 by time T and another transaction tx_2 after time T, then tx_1 will be committed before tx_2.

Another proposal, Pomp$\bar{e}$, decouples the transaction ordering process from the agreement process in order to obtain an alternative version of fairness [109]. The replicas involved in consensus express their preferences regarding the ordering of transactions, and given these sets of preferences, some possible total orderings are not considered valid. Specifically, if replicas base their ordering preferences on the time they first see a transaction, then the protocol guarantees the following: if the lowest timestamp that any honest replica assigns to tx_2 is higher than the highest timestamp that any honest replica assigns to tx_1, then tx_1 will precede tx_2 in the ledger. Unlike the order fairness property from [104], this ordering property is expressed based on the preferences of honest replicas rather than some fraction of all of the replicas, which may include Byzantine ones. This restriction results in protocols with improved fault tolerance compared to the Aequitas protocols, which require at least $4f+1$ replicas in order to tolerate f failures.

In addition to providing some notion of fairness in transaction ordering, other possible notions of fairness can exist in permissioned systems. For instance, protocols with leaders tend to have an uneven distribution of effort for replicas since the leader does a disproportionate amount of work. Rotating leaders or using leaderless protocols can help. Along these lines, [110] proposes a protocol designed to fairly balance the processing load based on past performance. It is similar to PBFT but with multiple simultaneous leaders. This system partitions client transactions among replicas according to each replica's processing ability, which prevents replicas from facing "unfairly" large resource burdens. Another possible notion of fairness arises if the protocol provides rewards to replicas, such as transaction fees. For example, FairLedger is a protocol that – assuming that otherwise-honest replicas are rational – ensures that each replica receives fair shares of the fees [111]. Along similar lines, [112] proposes the notion of *strongly fair validity*: if n replicas are involved in an instance of Byzantine agreement, then the probability that a particular replica's proposal is accepted by the honest replicas is lower-bounded by $\frac{1}{n} - \varepsilon$, where ε is negligible. *Weakly fair validity* captures the same idea but over repeated blocks, whereas strongly fair validity is with respect to a single-shot BA instance.

7.2. Accountability Against Malicious Replicas

Another useful property – and one that is occasionally used to make proof-of-stake protocols more robust – is accountability. In an *accountable* BA protocol, honest nodes that are not in agreement can exchange sufficient information to provably identify at least $\frac{n}{3}$ malicious nodes if $f \geq \frac{n}{3}$. This property relies on the idea that malicious equivocation can be detected due to the existence of two signatures from the same key on conflicting messages. When misbehavior can be detected by honest parties and individual responsibility can be assigned, it may be possible to relax some security assumptions while maintaining system security [113]. Accountability can also be used to help make BFT protocols more secure when some participants are rational rather than altruistic [114].

A protocol that provides this is Polygraph, which increases the communication complexity of the BA protocol it builds off of by a linear factor [115]. Polygraph relies on the leaderless Democratic BFT protocol proposed in [116] and used in the Red Belly Blockchain project [117]. It is also used in the Long-Lasting Blockchain (LLB) protocol in order to tolerate considerably more than the typical f Byzantine failures [118]. When there are more than f Byzantine replicas, a fork can be created and detected via conflicting signed messages. When LLB detects a fork, there is a recovery procedure that merges the conflict instead of discarding one of the conflicting blocks.

Reference [119] shows how to provide an alternative notion of accountability in a variety of BFT algorithms, including PBFT and HotStuff. Specifically, if $f < \frac{n}{3}$ is the maximum number of Byzantine faults that the protocol tolerates but $t > f$ Byzantine faults occur, then this notion of accountability guarantees that the protocol can detect at least $f+1$ Byzantine faults, where $2f+1-t$ honest replicas can testify to that effect, and proof of malfeasance only requires communicating with one of those honest replicas.

7.3. Specially Designated Roles for Replicas

Instead of requiring every replica to perform the same actions throughout a protocol execution, some protocols may assign special roles to some replicas in order to enhance performance. A simple example of this was seen in the HotStuff protocol (see Section 5.3), where replicas sent their signature shares directly to the leader for aggregation instead of multicasting them to every replica. The use of threshold signatures in this way is fairly common because it can reduce the communication complexity from quadratic to linear.

This technique is also employed in SBFT, where the aggregating party is called a "collector" [120]. In addition, SBFT uses the optimistic fast path from Zyzzyva, where the client can be viewed as the collector (see Section 4.3), but makes it more resilient by adding redundancy such that more than c faulty replicas are required to leave the fast path. A trade-off is that including this redundancy requires more replicas to maintain the same degree of fault tolerance ($n \geq 3f+2c+1$). SBFT recommends $c \leq \frac{f}{8}$ and uses $c+1$ collectors who rotate in round-robin fashion. SBFT further reduces client communication via threshold signatures. Instead of waiting for $f+1$ replies from replicas, an "execution collector" gathers the replies into a single signature over the result to send to the client.

Another technique, employed in the Proteus protocol, is to elect a subset of c replicas with $c << n$ as a "root committee," which is responsible for executing a BFT algorithm among themselves [121]. The block proposed by the committee is then validated by the remainder of the replicas, and if valid, the replicas sign it and return it to the root committee. When the root committee sees $2f+1$ signatures, they commit the block and send the signatures to the remainder of the replicas who will then commit as well. Proteus ensures stable performance regardless of the number of failures. View changes do not just change the leader but rather the full committee of c replicas. Compared to the typical $O(n^2)$ communication complexity of PBFT (and $O(n^4)$ for view changes), Proteus has a complexity of $O(c^2+cn)$ for normal and view change modes. The root committee tolerates up to $\frac{2c}{3}$ failures because the remainder of the replicas initiate a view change of the committee if the committee fails to propose a block.

7.4. Deterministic Longest Chain Protocols

Another style of BFT algorithm is the deterministic, longest-chain protocol, such as the "proof of authority" (PoA) algorithms Aura and Clique, which have been used for permissioned deployments of Ethereum. This kind of protocol favors availability over consistency during network partitions, not unlike Bitcoin and other "longest chain" protocols (see Section 10 on Nakamoto Consensus) [122]. This means that the chain can fork temporarily. Unlike with Bitcoin, the protocols mentioned here are deterministic, so the schedule of block proposers is known in advance. In other BFT protocols, leader election and voting are separate processes, but deterministic longest chain protocols combine them. When a replica is elected leader, it has the opportunity to vote, and honest replicas vote on the most "popular" ledger seen so far.

Aura assumes that replicas have synchronized clocks, and the next leader is elected at regular intervals. In Aura, the leader broadcasts a signed block to the other replicas, the leader is rotated in round-robin style, and future leaders build on the longest chain that they have seen. A block is considered final after a sufficient number of the replicas sign blocks on the same chain extending it. This uses only two rounds of communication as opposed to PBFT's three. Clique allows multiple simultaneous block proposers, resolves forks using the GHOST protocol (see Section 11.2), and only requires a single round of communication to commit. Early variants of both of these algorithms were subject to various attacks, including the "cloning" attack that allowed double-spending during network partitions [123]. Note that using GHOST as the fork-choice rule in a permissioned system may be dangerous due to the *balance attack*, where an adversary keeps the network partitioned and prevents transactions from being finalized by keeping the two partitioned blockchains at similar lengths for a period of time [124, 125].

In [126], the security of a variant of Aura is proven when $n \geq 3f+1$ in synchronous networks while simultaneously pointing out a vulnerability in an existing implementation. The attack prevents consistency by having the network constantly switch back and forth between equal-length chains. Prior to this analysis, it had been wrongly believed that the protocol was secure as long as the majority was honest. Randomized variants of longest chain protocols like Nakamoto Consensus have a better security margin (honest majority) and better confirmation times, but deterministic variants are secure against a more powerful adversary capable of "after-the-fact message removal," where "the adversary can observe what an honest node i wants to send in some round r, adaptively corrupt node i, erase the message it originally wanted to send, and then insert arbitrary corrupt messages on behalf of node i in round r" [126].

Ouroboros-BFT is a deterministic longest chain protocol similar to Aura and secure when $n \geq 3f+1$ [127]. It is synchronous and, thus, not optimally resilient for synchronous protocols (though it is optimally resilient for deterministic longest chain protocols). In the *covert* setting, where an adversary does not want to create evidence of their misbehavior (the accountability property discussed in Section 7.2 is relevant here), Ouroboros-BFT can tolerate $n \geq 2f+1$. The clock advances in slots every few seconds. Each time the clock advances to a new slot, replicas receive some transactions and blockchain candidates from the network and 1) update their mempool with the new transactions, 2) update their local preferred blockchain via the longest chain rule, and 3) check whether they are the current round-robin slot leader. If so, they extend their longest chain with a new block and diffuse it to the other replicas. A block is finalized when it has a slot timestamp more than $3f+1$ slots in the past.

Ouroboros-BFT has a few advantages and trade-offs compared to algorithms like PBFT and Zyzzyva. Ouroboros-BFT is simpler in that replicas perform the same steps in each slot independently of their view, whereas replicas in PBFT execute different steps depending on their current state. Synchrony assumptions differ, and PBFT maintains consistency with unbounded delays. When the network delay bound Δ is violated in Ouroboros-BFT, the chain forks, but replicas will regain a consistent view when the delay bound is respected again. Ouroboros-BFT provides speculative execution immediately, while the longest chain mechanism orders transactions over time, and communication complexity is optimistically $\Theta(n)$ per round and $\Theta(nf)$ in the worst case. Transactions have guaranteed liveness after $5f+2$ rounds, but a speculative outcome is produced in only two rounds. Zyzzyva has the same optimistic performance but requires the client to be actively involved in consensus, unlike PBFT and Ouroboros-BFT.

7.5. Flexible BFT

Most BFT algorithms provide safety and liveness for all users under certain conditions, such as a synchronous network with the majority of replicas being honest. Flexible BFT separates the fault model from the protocol design itself, allowing flexible beliefs about the network and number of faults. A client may specify a fault threshold and a maximum message delay, and safety and liveness will be maintained for any client with correct beliefs (and any two clients with correct beliefs will be in agreement with each other) [128]. Replicas commit transactions the way other BFT algorithms do: only clients commit, and they may do so at different times. The flexibility provided here is akin to the recipient deciding how many confirmations to wait for in Bitcoin. A $10 million transaction deserves a lengthier confirmation period than a $10 one. Further, clients may update their beliefs based on observation, such that if they notice more votes on conflicting values or lengthier message delays, they may become more conservative.

Flexible BFT also introduces a new *alive-but-corrupt* (a-b-c) fault model, where an adversary will try to violate the safety of the protocol but, failing that, will not try to disrupt liveness. This model may be justified when violating safety could reward an adversary with more money (e.g., via double-spending) but violating liveness may not (because keeping the service running may allow the adversary to collect transaction fees). This relaxation of the fault model allows for protocols that remain secure despite a combination of a-b-c and Byzantine faults in excess of $\frac{n}{3}$ under partial synchrony and $\frac{n}{2}$ under synchrony. To achieve these properties, Flexible BFT uses two key techniques:

1. Replicas run a partially synchronous protocol, but clients assume synchrony bounds for committing, which allows flexible assumptions regarding Δ. That is, the timing assumptions for replicas and for clients are distinct.
2. Replicas will use a particular quorum size while executing the protocol, but clients choose their own quorum sizes before committing. This allows clients to have divergent beliefs about the security threshold.

The use of quorums in proving the security of BFT protocols is discussed in Section 4.2. *Quorum intersection* both within and across views can prove safety by demonstrating that at least one honest replica would have needed to misbehave to violate safety. *Quorum availability* can be used to prove liveness by showing that a sufficiently large quorum contains no Byzantine replicas, and thus there are enough honest replicas to respond to an honest leader.

Flexible BFT separates out the quorums needed for different parts of the algorithm: locking on a value or forming a certificate requires q_{lck}, while unlocking requires q_{ulck}. The quorum that clients need for learning certificate uniqueness is q_{unq}, and the quorum required for safely committing is q_{cmt}. Clients require a q_{unq} fraction of votes in the first round and a q_{cmt} fraction of votes in the second round to commit.

Flexible BFT can ensure quorum intersection within a view by ensuring that every q_{lck} quorum intersects with every q_{unq} quorum at a minimum of one honest replica, which requires that the fraction of faulty replicas is less than $q_{lck} + q_{unq} - 1$. This guarantees that if a client commits a value, it is the only value with a certificate in this view. To ensure quorum intersection across views, every q_{ulck} quorum must intersect with every q_{cmt} quorum at a minimum of one honest replica, which requires that the fraction of faulty replicas to be less than $q_{ulck} + q_{cmt} - 1$. This guarantees that when a client commits a value, replicas that have locked on that value will not later unlock it. To ensure quorum availability within a view and thus liveness, the fraction of Byzantine replicas must be less than $1 - max(q_{unq}, q_{cmt}, q_{lck}, q_{ulck})$.

Balanced quorum sizes are optimal, such that $q_{lck} = q_{ulck} = q_r$ and $q_{unq} = q_{cmt} = q_c$. Additionally, $q_c \geq q_r$, because the q_r votes that replicas use to lock can also be used in the q_{cmt} quorums by clients. Here, q_r is the quorum that replicas need to lock a value (q_r is chosen by the system designer), and q_c is the quorum that clients need for safety (chosen by clients, along with Δ). Taken together, *flexible quorum intersection* holds when the fraction of faulty replicas is $< q_c + q_r - 1$, and *flexible quorum availability* holds when the fraction of Byzantine replicas is $\leq 1 - q_c$. For example, if $q_r = 0.7$ and $q_c = 0.75$, a client can maintain security when the fraction of Byzantine replicas is 0.25 and the fraction of a-b-c replicas is up to 0.2.

Other protocols have similar goals to Flexible BFT in that they attempt to accommodate a wider variety of assumptions simultaneously. For example, the protocols in Section 8 can allow replicas with different trust assumptions to maintain agreement. The Heterogeneous Paxos protocol allows different clients to set their own mixed failure tolerances for crash and Byzantine faults using differently trusted sets of replicas, essentially combining some of the benefits of Flexible BFT and the protocols from Section 8 [129]. The Highway protocol allows flexible assumptions regarding the number and types of faulty replicas and then uses these assumptions to establish different confidence thresholds for the finality of blocks [130]. Other protocols provide similar flexibility to Flexible BFT, but rather than using PBFT as the basis for the algorithm, ones with better communication complexity like HotStuff and Streamlet are used instead [131].

7.6. View Change Algorithms

In partially synchronous permissioned consensus protocols with leaders, *view changes* are the mechanism used to maintain liveness despite a malicious leader. A *view* can be considered a phase of the protocol where a particular replica acts as the leader and is generally represented as a monotonically increasing integer. A view change algorithm, sometimes called a *pacemaker* or *synchronizer*, has every replica start at view zero. The algorithm allows replicas to signal that they wish to advance to the next view but only actually advances the replica to a new view when the synchronizer emits a notification to the higher-level consensus protocol.

The challenge of view synchronization is that during good periods, progress can be maintained by the f Byzantine replicas combined with the lowest latency group of $f+1$ honest replicas. In this case, it is possible that only the quickest $f+1$ honest replicas recognize that a block has been agreed upon and advance to the next round of the consensus protocol, while the f slowest-but-honest replicas fall behind. Later, if the f Byzantine replicas stop behaving as though they were honest, the view change mechanism is required in order to get the f slow replicas to "catch up" to the same view as the $f+1$ quicker ones. The *Byzantine view synchronization problem* is solved by algorithms with the following two properties [132]:

1. View Synchronization: All honest nodes must execute the same view with an honest leader for a sufficiently long time, and there must be an infinite number of views with an honest leader.
2. Synchronization Validity: A synchronizer will only signal a new view if an honest node wished to advance to it.

A naive approach to view synchronization is to simply double the duration of each view, which ensures that there will be a sufficiently long period where all honest nodes share the same view. This has the advantage of requiring no communication between parties but is clearly impractical due to the unbounded latency that it introduces into the system. A more practical approach is the broadcast-based view change mechanism used in PBFT (see Section 4.1), which has high communication complexity but $O(\Delta)$ expected latency and $O(f\Delta)$ worst-case latency. Using this approach, when a replica sees messages suggesting that $f+1$ replicas wish to enter the same view, it relays its own wish to advance to that view using reliable broadcast.

These algorithms leave room for improvement, and several works have proposed new view synchronizers with better performance [132–134]. For example, the Cogsworth synchronizer matches the latency and communication complexity of the broadcast variant but, in some scenarios, can improve the communication complexity by a linear factor [132]. Instead of having every replica multicast messages to every other replica, they send a message directly to the leader of the view to which the replica wishes to advance. An honest leader then broadcasts a single message with an aggregated threshold signature with only linear communication complexity. If the leader of a view is Byzantine instead, then replicas may need to time out and attempt each of the $f+1$ subsequent leaders until they find an honest leader who will help relay the aggregated signature.

The Cogsworth protocol involves four types of messages, including two sent from replicas to leaders and another two sent from leaders to the rest of the replicas. Replicas send to prospective leaders $(WISH, v)$ and $(VOTE, v)$ messages when they want to move to the next step of the protocol, where v is the view number in question. The two that leaders send are "time certificate" and "quorum certificate" messages, each with one aggregated threshold signature: the leader sends (TC, v) upon receiving $f+1$ $(WISH, v)$ messages or sends (QC, v) upon receiving $2f+1$ $(VOTE, v)$ messages. When a replica wishes to advance to another view v, it sends $(WISH, v)$ to the leader of view v. If the leader of view v receives $f+1$ of these messages, they broadcast (TC, v). However, if 2Δ time passes after the replica sends its first $(WISH, v)$ without receiving a corresponding (TC, v) message, it sends $(WISH, v)$ to the leader of view $v+1$. The replica continues to wait in 2Δ increments, sending $(WISH, v)$ to the leader of view $v+2$ until it eventually receives a (TC, v) message. Upon receiving (TC, v), replicas send $(VOTE, v)$ to the leader of view v, even if they had not already sent a $(WISH, v)$ message. The replica waits 2Δ time to receive a (QC, v) message and contacts subsequent leaders as above if time runs out. In this case, the replica sends leaders both $(VOTE, v)$ and (TC, v). Replicas enter view v immediately upon receiving (QC, v) from a leader.

A follow-up work proposed an alternative view change algorithm built off of Cogsworth [133]. One issue with Cogsworth is that its expected linear communication complexity with benign failures becomes quadratic with Byzantine failures in the average case. The algorithm in [133] maintains an expected linear message complexity, even with Byzantine failures. To achieve this, the algorithm modifies Cogsworth by adding another phase to the algorithm and having replicas' signed messages to leaders include the identity of the leader it is intended for.

One problem with both of the above algorithms is that they rely on a modified variant of partial synchrony where there is no clock drift, and messages sent before GST cannot be lost and will necessarily arrive by time GST + Δ. This is potentially problematic because prior to GST, if clocks diverge, then the replicas' notion of the duration of a view can become desynchronized. Further, messages that may be needed in order to get replicas to be in the same view can be lost. Worse, the above algorithms require potentially unbounded memory space in order to store each $(WISH, v)$ message that falls below the required threshold for relaying, as well as maintaining a copy of every message it sends in order to enable it to be sent again. The FastSync synchronizer was designed to work under true partial synchrony with unbounded clock drift and the possibility of message loss before GST while running in bounded space [134]. FastSync works almost exactly like Bracha's broadcast (see Section 1.2), but replicas are not constrained to only acting on identical messages. Instead, replicas maintain only the highest $(WISH, v)$ message received from each process and can act on sets of $WISH$ messages for non-identical views.

One of the risks of the leader-based view-by-view progression of partially synchronous protocols is that leaders can be targeted in adaptive denial-of-service attacks in order to induce asynchrony and eliminate liveness. A leader-based view abstraction was designed in [135], which takes any view-by-view consensus protocol, wraps it in an API, and builds an asynchronous SMR system from it. This reduces performance in optimistic cases while dramatically improving liveness while under attack or while poor networking conditions hold. Separately, an asynchronous view change algorithm has been proposed as a replacement for the synchronizers discussed above. When combined with HotStuff, the resulting SMR system has linear communication complexity while the network is synchronous, quadratic complexity when the network is asynchronous, and always maintains liveness [136].

Strongly related to the idea of view changes is the ability to dynamically reconfigure consensus committees, such that replicas may join or leave the system over time. A variety of proposals exist that can provide such a functionality [137–142]. Reconfiguration can even be bundled with accountability (described in Section 7.2) in order to immediately change the committee upon detecting Byzantine behavior [142]. A system that is run by a single organization may not need to dynamically reconfigure the consensus committee, but when the replicated state machine is operated by a consortium of different organizations or individuals, the ability to add new members and remove others becomes more important.

To enable reconfiguration, the initial consortium members and their public keys are stored in the system's genesis block. In the scheme presented in [137], transactions that reconfigure the committee are included in separate *reconfiguration blocks*. Each normal block (the ones that execute client-issued transactions) includes a pointer to the previous reconfiguration block in the chain, so verifiers have access to the public keys needed to validate the signatures over proposed blocks. The primary security challenge while dealing with reconfiguration is preventing faulty replicas active in previous views from causing problems in later ones, as shown in Figure 15. The attack is analogous to long-range attacks against proof of stake, which is discussed in Section 12.1.2.

One way to resolve this problem is to separate a replica's "permanent" key pair from their "consensus" key pair, which is what is used for block signing. For each new configuration that a replica is involved in, it generates a new consensus key pair and discards the prior one (discarding the old key pair can be done using the "forgetting" scheme described in [143]). Securely discarding the old consensus key pair ensures that the faulty replica cannot recover the key and sign blocks corresponding to an old configuration should they become corrupted after a reconfiguration (as is done by replicas B and C in Figure 15). Reconfiguration blocks store the identities of the replicas and each of their consensus public keys in the new configuration.

For a new replica to join the system, it first asks the current replicas for permission. The current replicas choose to accept or reject the new replica, and if they accept, they send a signed reply message that includes the replica's intended consensus public key for the next configuration. Upon receiving $n-f$ signed acceptance messages, the prospective new

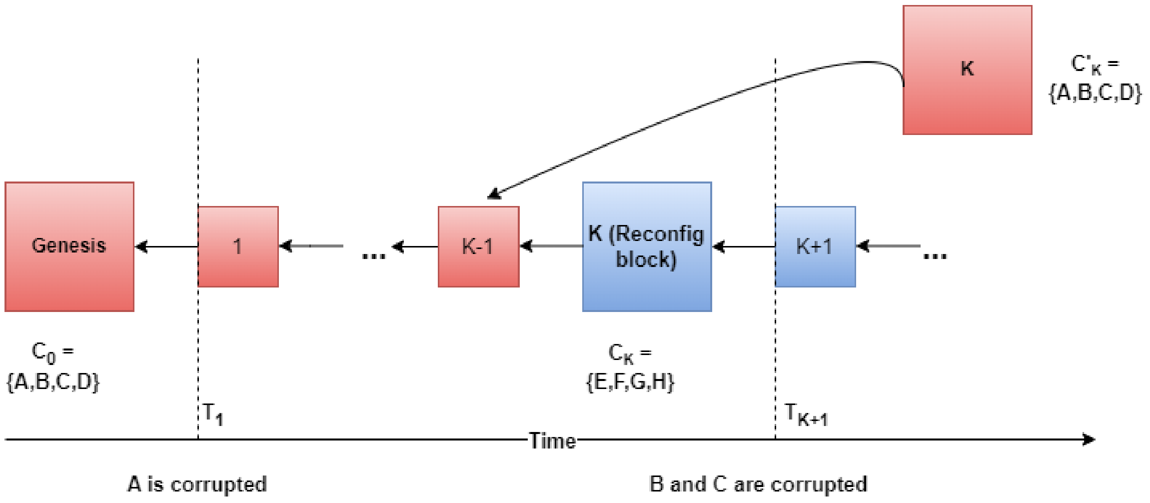


Fig. 15. Committee reconfiguration attack. While the threshold of faulty replicas is respected in each view, replicas A, B, and C work together to create a fork by extending the blockchain without the reconfiguration block at height K. This is possible even though none of the original replicas remain in the committee. If a client, aware of the initial configuration C_0, is offline during reconfiguration and then reconnects, they will contact the committee members from C_0. If this happens at any time after T_{K+1}, the corrupted quorum will convince the client to follow the wrong chain starting from height K.

replica creates a reconfiguration transaction that bundles these signatures together. When this transaction is included in a reconfiguration block and executed, the new replica can begin participating. Replicas can leave the system by themselves or by being kicked out by the rest of the replicas. If leaving by choice, the exiting replica gathers consensus public keys from existing replicas for a new configuration, bundles them together, and issues a reconfiguration transaction. If the replica is being kicked out, then existing replicas issue special reconfiguration transactions that ask the network to remove the target replica and provide a new consensus public key for the following configuration. Upon seeing $n-f$ of these transactions that remove the same replica, a new reconfiguration is generated without the replica.

8. Localizing Trust Over Incomplete Networks With Open Membership

The consensus protocols described in Sections 4 through 7 require that all n replicas in the system be fully aware of each others' identities. Further, they must all trust each other equally despite real-world relationships that may have varying levels of trust. Prior works have explored how to loosen restrictions on the knowledge of other replicas when networks are not fully connected [144, 145]. However, these BFT-CUP (Consensus with Unknown Participants) protocols still require that the unknown replicas be equally trusted as the known ones; that a fixed set of consensus replicas exist, each with a unique and

Sybil-resistant ID; and that all replicas are aware of the maximum failure threshold f. It is also possible to design permissioned agreement algorithms that are unaware of the precise values of n and f but maintain optimal resiliency [146]. Another line of work introduces asymmetric trust assumptions, where the replicas in the system do not adhere to a single global trust assumption [147, 148]. In these systems, each replica can choose which combinations of other processes it trusts and which may be considered faulty.

The protocols described later in this section combine some of the benefits of these systems. The key feature of these Federated Byzantine Agreement Systems (FBAS), sometimes called Federated Byzantine Quorum Systems (FBQS), is that each replica chooses its own group or quorum of trusted replicas to believe without needing to be aware of the existence of all other replicas or trusting them to the same degree. These systems represent a middle ground between permissioned and permissionless, though in practice, they are far closer to permissioned networks in both operation and trust levels. Section 4.2 contains some background information on Byzantine quorums.

8.1. Stellar

The Stellar Consensus Protocol (SCP) was introduced by Mazieres in [149] and further described by the Stellar Development Foundation in [150]. A generalization of Stellar's quorum system with security proofs is provided in [151].

8.1.1. FBAS Background

Stellar and similar protocols allow each user of the system to unilaterally define their own sets of trusted replicas, called *quorum slices*. A single organization may have multiple slices, any one of which suffices to convince them of a statement in case the others fail. A replica should choose their quorum slices such that they believe that

- If every member of a slice agrees about the state of the system, then they are correct about the state, and
- At least one of its slices will be available to provide information about the state of the system in a timely manner.

Let S be the set of replicas from which a set of messages originated. Every replica specifies its quorum slices in every message it sends, and it considers the set of messages to have reached the quorum threshold when every member of S has a slice included in S. A quorum is a non-empty set S of replicas that includes at least one quorum slice of each non-faulty member. If S is unanimous, the agreement requirements for all of its members are satisfied.

Two honest replicas v_1 and v_2 are considered *intertwined* when every quorum of v_1 intersects every quorum of v_2 in at least one honest replica. In an FBAS, agreement is only ensured between intertwined replicas. A set of replicas I is *intact* if I is a quorum whose replicas are uniformly honest, such that every pair of members of I is intertwined even

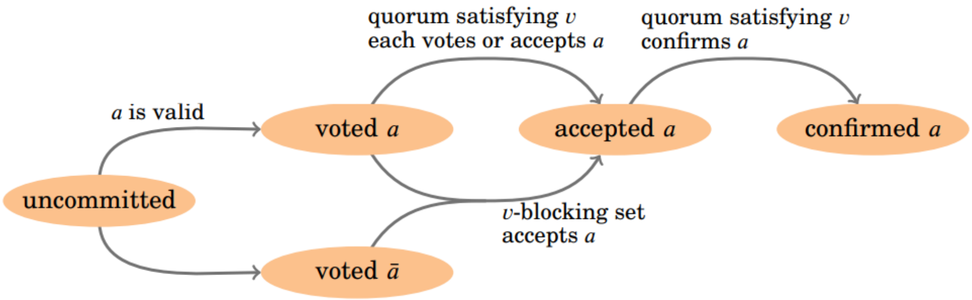


Fig. 16. Federated voting stages. [149]

if every replica outside of I is faulty. An intact set I cannot be harmed by the actions of non-intact replicas, and the union of two intact sets that intersect is an intact set. Stated differently, intact sets are partitions of the honest replicas, where each partition maintains safety and liveness under certain conditions but where different partitions may have divergent outputs.

A *v-blocking set* is a set of replicas that intersects every quorum slice of v. A v-blocking set B of faulty replicas can block progress by replica v because liveness requires that v has at least one quorum slice comprised solely of honest replicas. If B unanimously votes for a value x, then v knows that either x is true or v is not intact. A full quorum is required in order for v to know that x will not be contradicted by intertwined replicas. This requirement adds an additional round of communication to Federated Byzantine Agreement protocols compared to standard BFT protocols. Ultimately, there are three levels of confidence that a replica can have regarding consensus on a particular value: uncommitted, *accepted* (i.e., safe to assume among intact replicas), and *confirmed* (i.e., safe to assume among intertwined replicas). If replicas broadcast the values that they accept and a full quorum accepts a value, these values will propagate through intact sets due to the *cascade theorem*:

> If I is an intact set, Q is a quorum of any member of I, and S is any superset of Q, then either $S \supseteq I$ or there is a member $v \in I$ such that $v \notin S$ and $I \cap S$ is v-blocking. [150]

8.1.2. Stellar Consensus Protocol (SCP)

The Stellar Consensus Protocol is a partially synchronous protocol, where each single-shot attempt at achieving consensus on a value is called a *ballot*, and ballots employ increasing timeouts in order to synchronize replicas on the same ballot. Ballots are typically called views in other protocols. For each ballot n, a process called *federated voting* occurs on both PREPARE and COMMIT statements for a value x. Federated voting is the main subprotocol that provides agreement, and its stages can be seen in Figure 16.

A replica v will vote for any valid statement x that does not contradict its other outstanding votes and accepted statements by broadcasting a signed vote message. It then accepts x

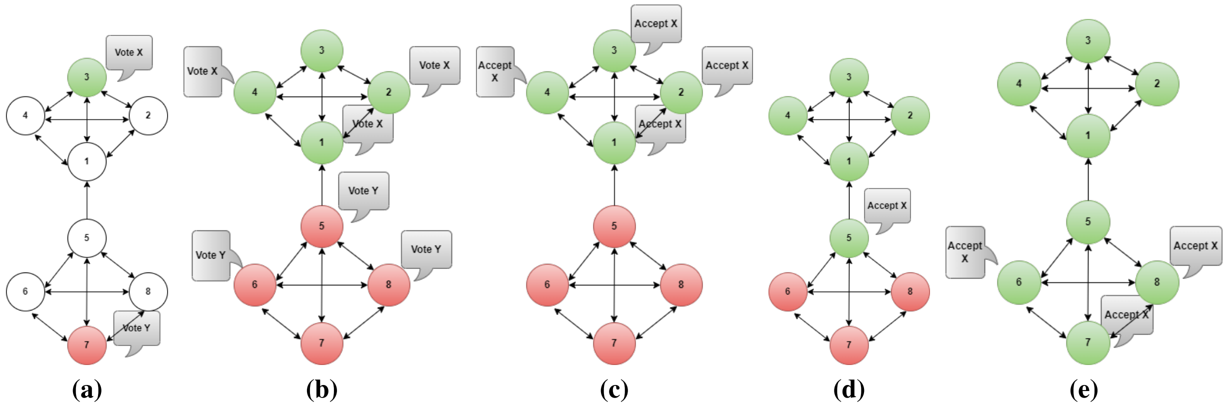


Fig. 17. Cascade effect in federated voting. Each replica has a single quorum slice indicated by arrows to members of the slice. All trust relationships are bidirectional with the exception of replica 5 including replica 1 in their quorum slice, but replica 1 does not reciprocate. (a) Contradictory statements X and Y are introduced. (b) Replicas vote for valid statements. (c) Replicas 1, 2, 3, and 4 accept X after their quorum $\{1,2,3,4\}$ unanimously votes for X. (d) Set $\{1\}$ is 5-blocking, so replica 5 accepts X and overrules its previous vote for Y. (e) Set $\{5\}$ is 6-, 7-, and 8-blocking, so replicas 6, 7, and 8 each accept X and overrule their previous votes for Y.

if x is consistent with other accepted statements and either 1) v is a member of a quorum where each replica either votes for x or accepts x, or 2) a v-blocking set accepts x even if v did not vote for x. In this case, if v had previously cast votes that contradicted x, those votes are overruled and forgotten. Replica v then broadcasts an accept message for x and confirms x when it is in a quorum that unanimously accepts x. Figure 17 shows an example execution of federated voting that demonstrates the cascading effect described by the cascade theorem.

Replicas will begin ballot n by initiating federated voting on a PREPARE(n,x) message. If a previous PREPARE message was successfully confirmed via federated voting, the replica will choose x as the confirmed PREPARE of the highest ballot. Otherwise, it sets x as the output of a *nomination* subprotocol. A replica attempts federated voting on COMMIT(n,x) if and only if the replica successfully confirms a PREPARE(n,x) in ballot n. If this succeeds, then the message is committed. However, if the replicas are unable to confirm a COMMIT message in the current ballot, they will employ increasing timeouts in order to synchronize on a particular ballot.

The nomination subprotocol involves federated voting over messages of the form NOMINATE(x). Once a replica confirms a single NOMINATE message, it stops voting to nominate new values, so the set of nominated values is finite. This process creates an evolving, deterministic combination of all values in confirmed NOMINATE messages. Intact replicas will eventually converge on the same set of values at some point arbitrarily late in the protocol execution. To reduce the number of values nominated, only a leader who

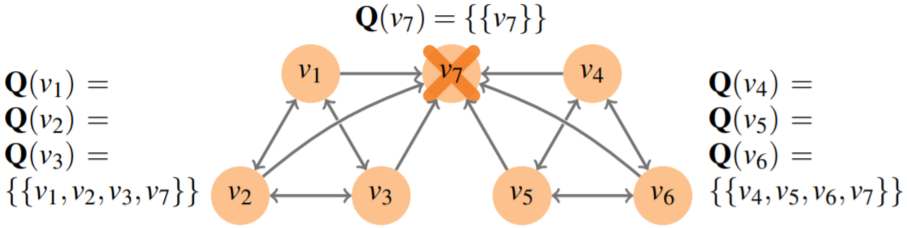


Fig. 18. FBAS Quorum intersection insufficient for safety. Quorums $\{v_1, v_2, v_3, v_7\}$, $\{v_4, v_5, v_6, v_7\}$, and $\{v_7\}$ intersect at v_7, but because v_7 is Byzantine, agreement is not guaranteed. [149]

has not already voted for a NOMINATE message may introduce a new x value. To tolerate failures, the set of leaders grows as timeouts occur.

The leader election mechanism employed in Stellar uses two hash functions, H_0 and H_1, where $H_i(m) = SHA256(i||b||r||m)$, b is the block number, and r is the leader election round number. Define $priority(v) = H_1(v)$. Define $weight(u,v) \in [0,1]$ as the fraction of replica u's quorum slices containing replica v. When u is selecting a new leader, it only consider its neighbors: $neighbors(u) = \{v \mid H_0(v) < 2^{256} * weight(u,v)\}$. Replica u starts with an empty set of leaders, and it adds the replica v in $neighbors(u)$ with the highest $priority(v)$ at each round. When the set is empty for a round, replica u instead adds the replica v with the lowest value of $\frac{H_0(v)}{weight(u,v)}$.

The normal-case operation of Stellar involves six messages: two to vote and accept a NOMINATE, two to accept and confirm a PREPARE, and two to accept and confirm a COMMIT.

8.1.3. SCP Security

Quorum intersection is a necessary but insufficient condition for the safety of the SCP protocol. This can be seen in Figure 18, where the replica shared across quorums is malicious. Instead, SCP requires that quorum intersection continues to hold, even after deleting all of the faulty replicas from the trust graph.

The federated voting procedure of SCP guarantees that no two members of an intertwined set confirm contradictory messages because the two quorums would share an honest replica that would not accept contradictory messages. However, a split vote can result in a statement becoming permanently stuck waiting for a quorum and not being confirmed. An intact set would not become stuck if a replica in it confirms a statement because replicas will vote on a value when a v-blocking set accepts it, and the cascade theorem ensures that this eventually spreads to the remainder of the intact set. Once the members of the intact set vote unanimously, confirmation is guaranteed.

To see why SCP itself maintains safety, consider an intertwined set S. The safety of federated voting ensures that, for a given ballot, a maximum of one value can be confirmed prepared by members of S. This implies that at most one value can be confirmed committed in a given ballot as well. Now assume that COMMIT(n,x) is confirmed. This implies that PREPARE(n,x) was also confirmed. PREPARE(n,x) would contradict any COMMIT(n',x') of different values from earlier ballots ($n' < n$), so federated voting guarantees that no other value was decided by members of S in any earlier ballot number. Therefore, SCP provides safety when quorum intersection holds in the quorums where misbehaving replicas are removed from the trust graph.

In addition to a sufficiently long period of network synchrony, liveness requires that a quorum exists and remains available even after deleting the Byzantine replicas. Specifically, a replica remains live only if it has at least one quorum slice comprised solely of honest replicas. The set of faulty replicas must not be v-blocking for any honest v in the system. As mentioned in Section 8.1.1, liveness can be guaranteed for intact sets.

If replicas do not start at the same time or have become desynchronized for any reason, timeouts alone will not suffice to achieve quorum availability within a ballot. Replicas begin their timers for ballot n only once they are part of a quorum where each replica is at ballot n or later, which prevents members of intact sets from staying too far ahead of the rest of the set. Additionally, if a replica v ever notices a v-blocking set at a later ballot, v will immediately skip to the lowest ballot such that this is no longer the case, regardless of timers. These mechanisms combined with the cascade theorem will help replicas that fall behind to catch up to the same ballot once the network experiences synchrony.

Assume that a ballot n is synchronous for a "long enough" time and that I is an intact set. By ballot $n+1$, all replicas in I will have confirmed the same (possibly empty) set of PREPARE messages, P. If $P = \emptyset$, the nomination subprotocol will converge on a value x. If P is not empty, let x be the value from the PREPARE message with the highest ballot number in P. In either case, replicas in I will perform federated voting on PREPARE$(n+1,x)$ in ballot $n+1$, so there will be a decision on x if ballot $n+1$ is also synchronous. This demonstrates that termination is guaranteed if the following conditions hold: the network is synchronous for two consecutive ballots, and the faulty members of honest replicas' quorum slices fail to interfere during those ballots.

Liveness, therefore, assumes that every replica in a quorum of a member of I must become synchronized or not send any messages at all for a sufficiently long period, which may require members of I to adjust their quorum slices. In an FBAS, replicas can unilaterally adjust their quorum slices at any time, making recovery from liveness failures substantially easier in an FBAS than in a typical closed BFT system. In a closed BFT system, consensus must be achieved on the replica reconfiguration events themselves, as described in Section 7.6, and this is especially challenging when the system has lost liveness.

Because quorum slices are user-configured, there is no guarantee that the emergent structure actually satisfies the security requirements regarding quorum intersection and availability. In fact, the actual Stellar network in January 2019 was sufficiently centralized that if a mere two replicas – both owned by the Stellar Foundation – were hacked or knocked offline, a liveness failure would have ensued, though improvements have been made since then [152]. Another line of work has shown that determining whether all quorums of a given FBAS intersect is actually an NP-complete problem, but there are some heuristic algorithms that can increase confidence that a particular configuration is secure [153–156].

Despite these heuristics, [155] points out that many of the likely ways that an FBAS would end up being configured revolve around a "top tier," or small group of replicas that are essential to providing safety and liveness to the whole system. The membership in this top tier cannot change without either the active involvement of existing top tier replicas or a loss of safety guarantees, which calls into question how "open" membership truly is in practice.

8.2. Ripple

The Ripple protocol was originally introduced in [157], but more up-to-date analyses are provided in [158, 159], which demonstrate that the original Ripple protocol has problems with both liveness and safety. Ripple-affiliated researchers have suggested a different algorithm – Cobalt (Section 8.3) – that could be used as an alternative for the network [160].

Not unlike Stellar, the Ripple protocol is designed to guarantee consistency even with only partial agreement on who participates in consensus. Each user who wants to participate defines a *unique node list* (UNL), which is the set of replicas that the user will trust for making decisions about the state of the network. Safety is determined by the intersection of UNLs between pairs of correct replicas. The original protocol authors believed that the minimum overlap necessary for security was 20%, but a later analysis suggested it was 40%, and finally [158] showed that it is actually more than 90%, which is a difficult condition to satisfy. This is the primary motivation for Cobalt, which only requires a 60% overlap. Further, if there is not universal agreement on the participants, the network can get stuck and lose liveness even with 99% UNL agreement and no faulty nodes, requiring manual intervention to continue making progress. In contrast, Cobalt can make progress during asynchrony.

In the following description of Ripple's consensus protocol, let the size of a UNL for replica P_i be n_i, and define a quorum, q_i, to be 80% of the UNL size. That is, replica P_i can tolerate no more than 20% of its UNL being faulty. Additionally, the term "ledger" is often used synonymously with "block" or "state" in the description. The consensus algorithm involves three steps: deliberation (proposing), validation (voting), and preferred branch (fork-choice rule). The algorithm is similar to GHOST (Section 11.2) or leaderless versions of longest-chain permissioned protocols (Section 7.4).

In the deliberation phase, replicas iteratively propose sets of transactions to the other replicas in their UNL over the course of several rounds. In round r of deliberation, a replica will only include transactions present in at least $threshold(r)$ of the recently received proposals from their UNL, where the thresholds increase from $0.5 \rightarrow 0.65 \rightarrow 0.70 \rightarrow 0.95$ as r

increases. The increasing thresholds are intended to prevent slow replicas from preventing consensus. When a replica sees a quorum q_i of its trusted nodes agree on the transaction set, it applies the transactions to its ledger L, broadcasts a *validation* (vote) $V_{L,i}$ on the set, and begins a new round of deliberation.

Replicas will only issue a validation on one block with a given sequence number. More formally, replica P_i only issues a validation $V_{L,i}$ for a block L if its height or sequence number $seq(L)$ is greater than that of any previous block validated by P_i. If the replica determines that it is not on the preferred branch, it will switch to the preferred one but will not issue validations until it catches up to the sequence number it was on prior to switching. In the validation phase, replicas listen for validations from other replicas in their UNL. If replica P_i sees a quorum q_i of validations for block L, then P_i sets the new fully validated tip to L.

The preferred branch phase is the chain-selection or fork-choice rule, and it is used to determine how to make progress when the network is not synchronous. The preferred branch selection algorithm only switches to a different branch when it knows that enough replicas have committed to that chain of blocks such that an alternative chain cannot have more support. Let $lastVals$ be the set of most recently validated ledgers (blocks). The three types of support used in the algorithm are:

1. **Tip support**: The number of trusted replicas whose most recently validated block is L

   $supp_{tip}(L) = |\{V_{L',i} \in lastVals : L = L'\}|$

2. **Branch support**: The number of trusted replicas whose most recently validated block is either L or descended from L

   $supp_{branch}(L) = supp_{tip}(L) + |\{V_{L',i} \in lastVals : L \in ancestors(L')\}|$,

   where $ancestors(L')$ traces back to the genesis block

3. **Uncommitted support** on sequence number s: The number of trusted replicas whose most recent validated block has either a sequence number lower than s or lower than that of the highest block L validation that the replica has personally broadcast

   $uncommitted(s) = |\{V_{L',i} \in lastVals : seq(L') < max(s, seq(L))\}|$

To find the preferred ledger, a replica walks down the blockchain starting from the block L that is the common ancestor of the most recently validated blocks. The replica then selects the child block $L' \in children(L)$ with the highest $supp_{branch}(L')$ that would still have the most support even if all $uncommitted(seq(L'))$ picked a conflicting fork. If no child of L satisfies this requirement, then L is the preferred ledger. However, if L' does exist, then the process is repeated on the children of L'. To break ties on forked blocks, the one with the higher hash value is selected. If this process leads to a block that is an ancestor of the replica's current working ledger, they keep that ledger as preferred, since they do not

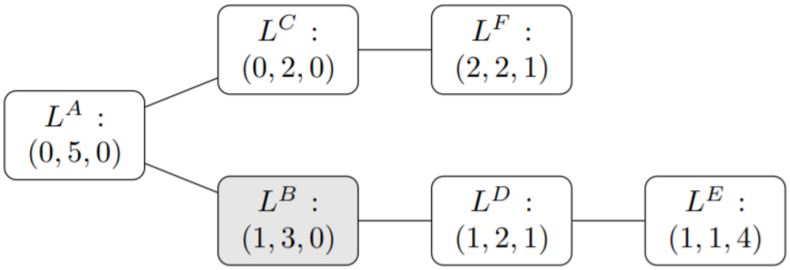


Fig. 19. Ripple "support" example. Each ledger is labeled with the tuple ($supp_{tip}$, $supp_{branch}$, $uncommitted$) from the view of a replica that last validated L^F and has five UNL members. Two trusted replicas had last validated L^F, and one each had last validated L^B, L^D, and L^E. The preferred branch is L^D. [158]

yet know that they are on the wrong branch. See Figure 19 for an example of the support definitions being used to choose a preferred branch.

8.3. Cobalt

Cobalt is another algorithm intended for use in open networks with individualized trust assumptions, and it is slated to be implemented in the Ripple network [160]. As mentioned in Section 8.2, Cobalt lowers the UNL overlap requirement from 90% to 60% while providing liveness in any network that satisfies this overlap bound between every pair of honest replicas. Further, the overlap condition for safety is *local*, so two replicas with sufficient overlap with each other cannot arrive at inconsistent ledger states, regardless of the overlaps between other pairs of nodes. That is, poorly configured replicas are unable to harm properly configured replicas. This makes evaluating whether the network is in a safe condition easier in comparison to Stellar and the original Ripple protocol, where evaluating whether a configuration is safe is an NP-complete problem. Unlike the earlier Ripple protocol and SCP, Cobalt maintains liveness in asynchronous networks. The Cobalt algorithm is similar to Bracha's broadcast (see Section 1.2) but generalized to incomplete networks and combined with a few other techniques.

8.3.1. Background

To generalize classic BFT algorithms and techniques to operate in a setting with incomplete networks, some additional concepts must be introduced. Define the *extended UNL* for replica P_i, denoted UNL_i^{∞}, to recursively include the UNLs from all honest replicas in UNL_i^{∞}. The extended UNL is P_i's view of the whole network and includes any replica that may have some impact on P_i. Similar to the idea of quorums in Ripple, replicas have a set of *essential subsets*, ES_i, such that $UNL_i = \bigcup_{E \in ES_i} E$. If $S \in ES_i$ for P_i, define $n_S = |S|$, and two parameters t_S (the maximum tolerable number of Byzantine replicas in S for safety), and q_S (the number of correct replicas needed to guarantee liveness) that satisfy the following:

1. $0 \le t_S, q_S \le n_S$.
2. $t_S < 2q_S - n_S$. This means that any two subsets of q_S replicas must intersect at an honest replica unless the number of Byzantine replicas exceeds the t_S threshold. This guarantees consistency.
3. $2t_S < q_S$. This ensures that liveness holds for replicas with $S \in ES_i$.

Replicas can configure q_S and t_S for each $S \in ES_i$ on an individual basis, and the above conditions hold when $n_S > 3t_S + 1$ and $q_S = n_S - t_S$.

Replicas P_i and P_j are considered *linked* if there is an essential subset $S \in ES_i \cap ES_j$ such that fewer than t_S replicas in S are Byzantine. P_i and P_j are *fully linked* if there is some essential subset $S \in ES_i \cap ES_j$ such that at least q_S replicas in S are honest, at most t_S replicas in S are Byzantine, and $t_S \le n_S - q_S$. Linkage is important for consistency while full linkage helps provide liveness. A replica is said to be *healthy* if it is honest and no more than $min\{t_S, n_S - q_S\}$ replicas in each of its essential subsets are not healthy. If a replica is healthy, then even the influence of Byzantine replicas cannot cause it to accept or broadcast arbitrary messages. Similarly, a replica is *unblocked* if it is healthy and at most $min\{t_S, n_S - q_S\}$ replicas in each of its essential subsets are not unblocked. Blocked nodes can be arbitrarily prevented from terminating. Replica P_i is *strongly connected* if every pair of healthy replicas in UNL_i^∞ are fully linked with each other and *weakly connected* if it is fully linked with every healthy replica in UNL_i^∞.

A replica P_i sees *strong support* for a message M if P_i receives M from q_S replicas in every essential subset $S \in ES_i$. Similarly, P_i sees *weak support* for a message M if P_i receives M from $t_S + 1$ replicas in some essential subset $S \in ES_i$. The security of Cobalt can be derived based on the following two principles, which suffice to translate normal BFT techniques that work in complete networks to ones that can operate securely in incomplete networks:

1. If two replicas are fully linked, then their essential subsets will overlap to the point where if one of the replicas sees strong support for a message, the other replica will eventually see weak support.
2. If two replicas are linked, then their essential subsets will overlap to the point where the replicas cannot simultaneously see strong support for messages that contradict each other.

The Cobalt protocol itself combines a number of subprotocols that are each adapted to work in incomplete networks. In particular, it uses an adapted form of reliable broadcast (RBC) and a multi-value Byzantine agreement (MVBA) that itself uses asynchronous binary Byzantine agreement (ABA). In essence, proposers distribute their proposals using a variant of reliable broadcast called democratic RBC (DRBC), which is described in detail in the next section, and then use MVBA to agree on the assignment of a single proposal for each slot. The ABA algorithm is adapted from Mostéfaoui's protocol described in Section 6.1.1. An extra messaging round is added to Cobalt's version in order to guarantee that the ABA's consistency holds as a local property, which is a significant advantage in open networks. The MVBA algorithm is out of scope for this document.

8.3.2. Broadcast in Incomplete Networks

The broadcast problem and Bracha's solution to it are described in 1.2. Cobalt generalizes the problem and solution to incomplete networks. A solution to the broadcast problem in this environment with designated sender B_i has the following properties:

1. **Consistency**: If an honest replica accepts a message M, no honest replica linked to it ever accepts an alternative message $M' \neq M$.
2. **Reliability**: If a replica is strongly connected and any healthy replica in its extended UNL accepts a message, then all unblocked replicas in the extended UNL eventually accept the message.
3. **Validity**: If B_i is honest and broadcasts a message M, then any healthy replica that accepts a message must accept M.
4. **Non-triviality**: If B_i is honest and can broadcast to every honest replica in the network, then every unblocked replica will eventually accept the message.

A solution to the democratic reliable broadcast problem has two additional properties that replace non-triviality, and allows replicas to elect to support or oppose a message.

1. **Democracy**: If a healthy and weakly connected replica accepts a message M, then there is an essential subset $S \in ES_i$ such that the majority of all honest replicas in S supported M.
2. **Censorship-resilience**: If B_i can broadcast to every honest replica in the network and all honest replicas support M, then every unblocked replica will eventually accept M.

The reliable broadcast protocol below begins by having the designated sender B_i multicast INIT(M) to everyone who will listen. Then all replicas P_j perform the following steps, where an empty parenthesis implies an arbitrary message.

1. If a replica receives an INIT(M) message directly from B_i, it multicasts ECHO(M) if it has not already sent ECHO().
2. If a replica sees weak support for ECHO(M), it multicasts ECHO(M) if it has not already sent ECHO().
3. If a replica sees strong support for ECHO(M), it multicasts READY(M) if it has not already sent READY().
4. If a replica sees weak support for READY(M), it multicasts READY(M) if it has not already sent READY().

5. If a replica sees strong support for READY(M), it accepts M.

This RBC protocol can be modified to convert it into a DRBC protocol by having replicas only send ECHO(M) messages if they actually support the proposal M. Even if a replica opposes the proposal, it must participate in the READY phase as normal.

The ECHO phase of the protocol is used to guarantee consistency, while the READY phase is used to provide reliability. To see why the protocol provides consistency, note that a replica only accepts a message M if it sees strong support for READY(M). If two honest replicas are linked and both accept a message, then by principle (2) above, the messages must be the same.

Proving reliability is trickier due to having much stronger network assumptions than consistency. If a replica P_k is strongly connected, and two healthy replicas P_i and P_j in its extended UNL send READY(M) and READY(M') messages, then $M = M'$. Steps 3 and 4 of the protocol have an honest P_i send READY(M) if it saw strong support for ECHO(M) or weak support for READY(M). For P_i to see weak support for READY(M), then a healthy replica in $UNL_i \subseteq UNL_k^\infty$ must have already sent READY(M) before P_i. This means that there was some healthy replica – say, $P_{i'}$ – that was first to send READY(M), and they did so because they saw strong support for ECHO(M). Therefore, one can assume that two healthy replicas exist, $P_{i'}, P_{j'} \in UNL_k^\infty$ such that $P_{i'}$ saw strong support for ECHO(M) and $P_{j'}$ saw strong support for ECHO(M'). By the definition of strongly connected, $P_{i'}$ and $P_{j'}$ are fully linked, so $M = M'$.

The insight from the previous paragraph can be used to prove reliability. Note that every pair of healthy replicas in UNL_k^∞ are fully linked. If $P_i \in UNL_k^\infty$ accepts M, then by principle (1), all unblocked replicas in UNL_k^∞ will see weak support for READY(M). The previous paragraph shows that a healthy replica in P_k's extended UNL cannot have already sent a READY(M') for $M' \neq M$. Step 4 of the protocol ensures that a healthy replica in this extended UNL will eventually send READY(M). If $P_j \in UNL_k^\infty$, then all healthy replicas in $UNL_j \subseteq UNL_k^\infty$ eventually send READY(M). If P_j is unblocked, it will eventually see strong support for READY(M) and thus accept M.

It is clear that the protocol provides validity: healthy replicas do not send ECHO(M) without having received either INIT(M) from the designated sender or ECHO(M) from another healthy replica. Since the sender is honest, it sends INIT(M), and no healthy replica will send an ECHO(M') with $M' \neq M$. Similar logic holds for the READY messages, implying that no healthy replica will see enough READY(M') messages to accept M'. Non-triviality is even simpler: every replica can receive INIT(M) from the designated sender, so all healthy replicas will send ECHO(M) and eventually READY(M), such that all unblocked replicas eventually accept M.

The proof of censorship-resilience for the DRBC variant is identical to the proof of non-triviality for the standard incomplete networks variant. To see why the democracy property holds, let P_k be a healthy and weakly connected replica that accepts M. Let

$P_i \in UNL_k^\infty$ be the first healthy replica to have sent READY(M). Then P_i saw strong support for ECHO(M). Weak connectivity implies that P_i and P_k are (fully) linked, so there exists an essential subset $S \in ES_k$ where at least $q_S - t_S$ honest replicas in S send ECHO(M) and no more than $n_S - q_S$ honest replicas in S did not send an ECHO(M). Because $t_S < 2q_S - n_S$ (the second assumption on the parameters, from Section 8.3.1), $q_S - t_S > q_S - (2q_S - n_S) = n_S - q_S$, so a majority of honest replicas in S supported M.

9. Proof of Work: The Basics

The concept of proof of work was first suggested by Dwork and Naor as a way to counter email spam by adding a small cost to sending emails [161]. The term "proof of work" was later coined by Jakobsson and Juels [162]. The proof-of-work scheme used in Bitcoin was inspired by Adam Back's "Hashcash" [163]. More recently, "resource burning" has been studied in general, which includes proof of work but also proof of space, which is discussed in Section 14.1 [164].

In the context of a replicated state machine, a proof-of-work consensus algorithm will consist of a specific proof-of-work puzzle, a difficulty adjustment algorithm, a fork-choice rule (and implicitly, a data structure to work on, like a blockchain or DAG), and an incentivization scheme. This section marks a shift in this document's discussion from permissioned to permissionless consensus algorithms and will discuss the basic functionality of proof of work and how it operates as a Sybil-resistance mechanism for distributed ledger systems.

9.1. Proof of Work and Sybil Resistance

The actual mechanics behind constructing a proof of work are quite simple. Some proof-of-work function must be defined, and a cryptographic hash function is typically used. A puzzle difficulty is chosen (in the consensus context, this is done with a difficulty adjustment algorithm, discussed in Section 9.2), that then determines the range of hash function outputs that would constitute a successful proof.

To find a proof of work, entities try to find partial hash collisions using the given function, difficulty target, and specified message format. The message needs to include anything that should be "tied" to the proof. For example, in the context of email spam, the message should include the contents of the email to be sent. For a proof-of-work cryptocurrency like Bitcoin, the proof must cover the entire contents of the block to be mined and the previous block hash (if a DAG is used instead of a blockchain, then multiple previous block hashes may be included). Given a proof-of-work function (H), a difficulty target to satisfy ($Target$), a previous block hash to mine on top of ($prevHash$), and a set of transactions to mine into a block (txs), a miner will repeatedly try different nonces until $H(prevHash||txs||nonce) < Target$.

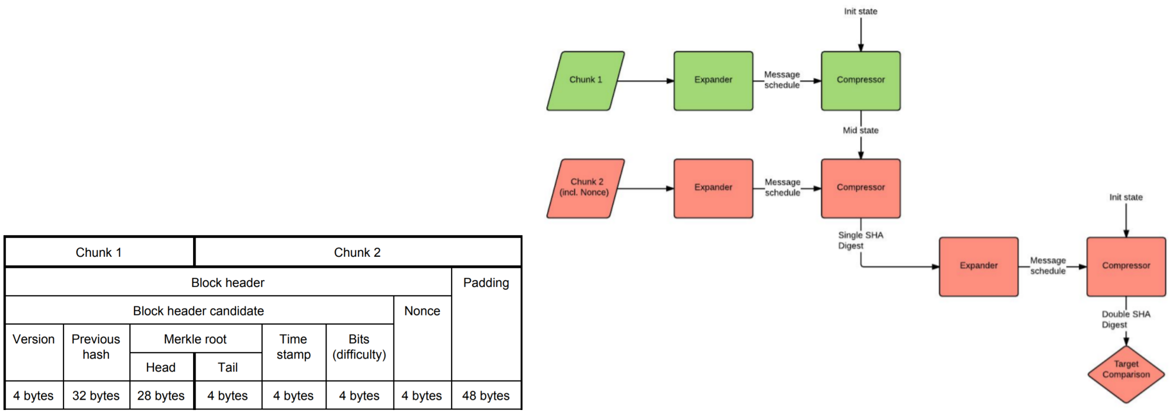
A good proof-of-work function has several important properties. First, the puzzle must be moderately hard, have this hardness be tunable, and be very fast to verify. Partial hash inversions are a good example of this, where the difficulty of the puzzle can be tuned by changing the *Target*, and verification is as fast as a single hash query. The function must also be *memoryless*, such that the time it takes to solve the puzzle does not depend on how much time has already elapsed or the history of attempts to solve the puzzle. That is, if a miner spends five minutes trying to mine a block, they are no closer to having found it than when they began. This is why verifiable delay functions by themselves are inadequate for proof of work. An adversary with a single processor with sequential processing speeds just slightly faster than honest parties will almost inevitably solve every puzzle first. Contrast this with a cryptographic hash function, where guessing nonces can be performed in parallel and where previous guesses do not bring the miner closer to finding a solution to the puzzle.

Finally, the most fundamentally important property of a good proof-of-work function is that there should be no strategies or shortcuts that aid in solving the puzzle. A cryptographic hash function (or at least one modeled as a random oracle) is a good example because there is no strategy better than arbitrarily guessing nonces. It is this property that ensures that any verified proof-of-work solution implies that the prover invested a sufficient amount of computational effort. This is what makes proofs of work useful tools in consensus: by imposing a cost on participants who want to send messages, parties running the consensus protocol have opportunities to synchronize their local views regarding the state of the system [165]. An adversary with a shortcut would be able to out-compete other miners and ultimately centralize the system under their control.

An example that vividly demonstrates the significance of potential shortcuts is AsicBoost, a technique that improves the efficiency of Bitcoin mining by about 20% [166]. Bitcoin's mining algorithm involves performing a double SHA-256 hashing of the 80-byte block header. SHA-256 operates on 64-byte chunks of the message at a time, so two chunks must be hashed. The Bitcoin block header – including how it is broken down into chunks for mining – is displayed in Figure 20a. The two chunks are processed in the manner shown in Figure 20b.

The outer loop of the mining process – shown in green in Figure 20b – preprocesses the first chunk using the expander and compressor functions of SHA-256 and results in an output called the *midstate*. The inner loop – shown in red in Figure 20b – preprocesses the second chunk analogously but then takes the midstate from the outer loop as input to a second round of SHA-256 expansion and compression. A *work item* in Bitcoin consists of the midstate generated from preprocessing the first chunk, as well as the tail, timestamp, and bits fields from the second chunk. The nonce is incremented in each run through the inner loop, requiring another two expansions and two compressions. The performance gains from AsicBoost come from reusing the first expansion of the second chunk across multiple work items. To do so, the miner needs multiple block header candidates that share the same tail, timestamp, and bits fields.

Notice how the Merkle root, which commits to every transaction included in the block, spans both chunks of the block header. The AsicBoost technique exploits collisions in



Chunk 1			Chunk 2				
Block header							Padding
Block header candidate						Nonce	
Version	Previous hash	Merkle root		Time stamp	Bits (difficulty)		
		Head	Tail				
4 bytes	32 bytes	28 bytes	4 bytes	4 bytes	4 bytes	4 bytes	48 bytes

(a) Bitcoin block header

(b) Bitcoin's Double-SHA256 mining.

Fig. 20. Bitcoin mining and AsicBoost. (a) Bitcoin block header, as used for mining. (b) Bitcoin's double SHA-256 mining. The outer loop is in green, while the inner loop is in red. [166]

the last four byte "tail" of the Merkle root. A miner employing the technique will find a set of Merkle roots that collide in these bytes by varying and/or rearranging the order of transactions included in the block (finding four byte collisions is not too challenging due to the birthday paradox). That is, the miner will have a large set of differing first chunks (and their corresponding midstates) that remain valid when paired with a particular second chunk. By reusing the current value of the second chunk with Merkle roots colliding in the tail, a miner can simply swap out midstates to their precomputed values while skipping the first expansion of the inner loop. As a result, the inner loop only has three large operations instead of four.

Suppose that a miner finds three Merkle roots where the final four bytes collide, and call the resulting midstates that arise from the first chunk A, B, and C. The resulting mining loop will begin with the miner setting the nonce to zero and using midstate A, requiring four large operations. The miner then keeps the same nonce but swaps in midstate B, which allows skipping the first expansion and only performing three operations. If this fails, computing using midstate C also only requires three operations. Then the miner sets the nonce to one and resets the midstate to A, computes the inner loop using four operations, and continues in this way, swapping midstates before incrementing the nonce.

One of the major reasons why the discovery of AsicBoost was a significant event in the Bitcoin community was due to patents. The technique itself was patented, but if the technique could not be used universally, it would provide a likely insurmountable mining advantage to the patent-holder. The AsicBoost patent is now held under the Blockchain Defensive Patent License, which obligates any participating entity to share their own mining-related patents. As a result, this is less likely to be an issue in the future. Further, the version of AsicBoost described above is no longer possible on Bitcoin due to the Segregated Witness soft fork performed in 2017. An overt version of AsicBoost that simply adjusts the version number in the first chunk instead of finding Merkle root tail collisions is still possible.

9.1.1. Mining Pools

Rewards for mining are provided to miners when new blocks are found. In Bitcoin, new blocks are found approximately every 10 minutes or 144 per day. Other networks produce blocks more frequently, though network latency places fundamental bounds on how quickly blocks can be produced (this latency issue is discussed in detail in Section 10.2.1). Because there are relatively few opportunities to mine blocks and thus collect rewards, it can take an exceedingly long time for small miners to find a block once there are a significant number of miners on the network. As a result, small miners could easily go out of business before mining a single block, and the variance in rewards for these miners is substantial.

This high variance in mining rewards is the primary motivation for *mining pools*, where a variety of hardware operators cooperate in mining and share the rewards among themselves. This allows smaller, more frequent mining payouts, which makes mining viable to a much wider variety of entities, including those with fewer resources. Pools operate by setting a difficulty level that is a small fraction of the difficulty of mining an actual block, and proofs of work at these lower difficulty levels are called *shares*. Participating miners submit shares to the pool operator in exchange for a portion of the pool's reward income (various reward schemes and attacks against them are discussed in Section 9.3). Because there are far more shares than blocks, there are many more opportunities for small miners to collect payouts.

Mining pool operators run full nodes and assemble block templates to send to the miners, providing each miner with a particular nonce range to search in. Miners themselves simply operate the hardware. They do not necessarily validate blocks or otherwise participate in the network. The protocol that pool servers use to communicate with miners is called Stratum, which lacks authentication and has resulted in some miners losing their payouts due to network-layer attacks [167].

It is a common misunderstanding that mining pools and miners are effectively the same thing, even though individual miners can easily switch pools at will. If a small set of mining pools are responsible for mining the vast majority of blocks, many people conclude that only a handful of entities control the network, oversimplifying the nature of "control." Under the current Stratum protocol, there is *some* truth to this. By assembling block templates and thus choosing which transactions to include in a block, mining pools can launch certain attacks or otherwise engage in nefarious behavior. This is one of the primary motivations for the ongoing development of Stratum v2, which eliminates these risks by letting miners select transactions while still pooling rewards [168]. Currently, due to the privileged position of mining pools as the entities that select transactions, the following behaviors are indeed possible:

- A mining pool can censor the inclusion of transactions that it does not like or that regulators tell them to. If a majority of the computational power of the network is not directed at censorship, however, it will only cause increased latency for transaction confirmation. Further, pools lose out on transaction fees by censoring, so even if the majority of the hash rate is malicious, there is an incentive for miners to deviate from the censoring cartel in order to capture more fee revenue.

- Pools can attempt chain reorganizations for double-spending. This is unlikely to succeed without a majority of the computational power of the network involved, and the miners themselves may have strong incentives not to go along with it. Additionally, as miners detect an attack in progress, they can migrate to another pool. If a successful double-spend would decrease the value of a miner's virtual assets or the hardware used to mine them, there is a significant incentive to deviate from the double-spending cartel.
- Pools can use their position to influence the rules of the network, using the miners as leverage. For example, if the pool operator believes that the rules of the network should change to allow higher throughput, they can mine empty blocks and allow a backlog of transactions to accumulate in order to increase fees and frustrate users. Pools can also direct hashpower to networks with alternative rules (but the same proof-of-work algorithm) without the consent of miners. For instance, miners who thought they were mining Bitcoin can be made to mine Bitcoin Cash instead. Similarly, one of the more common mechanisms employed in changing the rules of the network is miner signaling: miners set a bit in their block headers to signify approval. Pools can thus signal on behalf of miners who may or may not agree. As with the other possible attacks, miners can switch pools to those more aligned with their values (including, perhaps, profitability) once they detect that their current pool is behaving in ways that they do not approve of. To be clear, neither mining pools nor miners can arbitrarily change the rules of the network, even with a majority of the hash rate. Blocks that are invalid under the "old" rules will be rejected and ignored by the rest of the network.

An interesting consequence of having only a handful of major mining pools operating simultaneously is that many pool operators are known, identified entities. The implications of this are mixed: it should be easier for pool operators to coordinate nefarious activity or be coerced into performing attacks, but they can also be held more accountable for misbehavior due to a desire to maintain their reputation. Again, miners themselves can easily switch pools, so a pool that acts against the best interests of its miners may not survive for long.

Even if a (small) majority mining cartel is formed, deviating from the cartel can be highly profitable both for pool operators and the miners themselves. If the deviation pushes the cartel's influence back under 50%, then participants in the cartel gain no rewards, and the deviating party (as well as those who never participated in the first place) temporarily face less competition and thus collect greater rewards. Any pool operator representing enough hash rate to reduce the cartel back to a minority has a strong incentive to be the one who deviates. If the cartel agreement is only between pools, miners within those pools are likely to switch to other pools once the attack is detected. The miners themselves face electricity and other operational costs to participate in the attack but get no benefit from it (while risking the value of their assets, as described above). As a result of these factors, an attack by one or more mining pools is likely to only succeed in the short term rather than establishing long-term control over the contents of the blockchain. There are many more miners than pools, and miners may operate anonymously, so organizing a majority cartel of miners is far more challenging in practice. For these reasons, it is a significant oversimplification to suggest that the network is controlled by very few entities.

Due to the perceived centralization of mining into pools, there have been efforts to remove or reduce their power, including the aforementioned Stratum v2 protocol. Another effort is P2Pool, a more distributed mining pool. P2Pool miners create their own blockchain that is composed of their lower difficulty shares, and the difficulty is set such that shares are expected to be found every 30 seconds (or 20 times more frequently than blocks). The P2Pool sharechain holds 8,640 shares at a time, and payouts are performed proportionally to a miner's fraction of those shares. To enforce this, miners verify that any shares they build off of include the appropriate payouts in the coinbase transaction. Unfortunately, P2Pool has not been able to gain a significant amount of hash rate, in large part because the latency problems that arise in proof of work are magnified substantially, resulting in a very high rate of stale shares.

A more extreme way to neutralize the power of mining pools is to design a proof-of-work algorithm that entirely precludes pools from existing profitably in the first place. Specifically, *non-outsourceable puzzles* have the property that if a pool operator is able to outsource work to a miner, the miner is capable of stealing the reward in a way that does not implicate themselves [169]. There are two major problems with this approach:

1. In many cases, it is possible to devise smart contracts where miners must submit collateral that can be seized by the pool operator should a miner steal the block reward [170]. This essentially recreates pools but with higher overhead and complexity.

2. By reducing the variance of payouts, pools provide a valuable service. Many small mining operations simply would not be viable in the first place if it were not for pools. If non-outsourceable puzzles successfully eliminated mining pools, it is quite likely that there will be far fewer miners overall, resulting in even more centralization of mining.

9.1.2. Hardware: ASICs and ASIC Resistance

One of the more significant debates among cryptocurrency enthusiasts is whether it is desirable for the proof-of-work algorithm to be mined with specialized hardware, such as custom-built application-specific integrated circuits (ASICs) or general-purpose hardware like CPUs and GPUs.

ASIC resistance is a property of a proof-of-work algorithm that stipulates that it would be challenging to build specialized hardware that is capable of solving proofs of work much more efficiently than general-purpose hardware. Proponents of ASIC resistance argue that the wider availability of CPUs and GPUs results in a far lower barrier to entry for small miners while providing a mechanism for hobbyists to acquire cryptocurrency without needing to identify themselves to a centralized exchange. Large segments of the population already possess general-purpose hardware or can acquire it without revealing their specific interest in cryptocurrency. Contrast this with ASICs, which tend to cost hundreds or thousands of dollars per machine and which leave no plausible deniability for their purpose. ASICs can be intercepted at national borders or otherwise stopped by hostile governments or shipping companies, and their manufacturers can be identified and targeted. Additionally, ASIC manufacturers themselves can become important players in the "politics" of the networks they produce hardware for, which can include advocating for controversial rule changes and selectively selling machines to preferred partners.

Many approaches have been used in attempts to gain the ASIC resistance property, but the overwhelming majority of them have failed within a few years. Once there is sufficient money at stake – that is, when a proof-of-work cryptocurrency becomes valuable enough – it seems that hardware manufacturers find a way to build an efficient ASIC for the algorithm. It is unclear to what extent long-term ASIC resistance is even possible in the first place. One of the more promising attempts at ASIC resistance is called RandomX, which has most prominently been used for the Monero cryptocurrency since November 2019 [171]. RandomX is designed to be mined on CPUs. The algorithm relies on a pseudorandom key that is periodically extracted from the blockchain. This key is used to help generate random programs in a very general, low-level instruction set where any random string is a valid program. This is translated into machine code and executed in a way that uses as many components of the CPU as possible before being hashed into a final result. This use of many CPU components should complicate the design of an ASIC, which is programmed to only execute a very specific task. Only time will tell if ASICs are eventually built for RandomX as well.

On the other side of the debate are those who feel that proof of work is more secure when specialized hardware is widely used for mining. In this way of thinking, it is good for the security of a given network if its proof-of-work algorithm is the dominant application of a particular piece of hardware. An ASIC-resistant algorithm is likely to have a huge quantity of (unused) *potential* hash rate and may be more likely to have the security assumptions of the network (e.g., honest majority of hash rate) be violated. Every CPU or GPU in the world could potentially be turned on to attack the network. This is not solely a concern for ASIC-resistant algorithms. If two cryptocurrencies are mined using the same proof-of-work algorithm (e.g., Bitcoin and Bitcoin Cash), the less valuable one will suffer a security deficit for the same reason. In addition, ASICs – by only being useful for a limited purpose – require miners to have a vested interest in the network. If the network is attacked, ASIC owners will lose out on their initial hardware investment, which cannot be repurposed except to mine an alternative cryptocurrency using the same algorithm that may also be attacked by the same hardware.

Because many believe that long-term ASIC resistance is futile, attempting to provide ASIC resistance merely imposes barriers to entry for hardware manufacturers. This makes it even easier for the market to become concentrated, perhaps with a single major manufacturer that can build ASICs and mine with them in secret. A related risk is that frequently hard-forking to eliminate ASICs from the network creates strong incentives for collusion between developers and hardware manufacturers and, thus, introduces a strong point of centralization around the development of the mining algorithm. A related concern is that the designer of an ASIC-resistant algorithm that will initially be executed mostly on CPUs can develop an optimized GPU implementation that they can run in secret, providing a significant mining advantage.

Some of these security claims have been investigated in the literature. For instance, when multiple cryptocurrencies use the same proof-of-work algorithm but one has much higher difficulty than the other, rational miners from the dominant chain may be incentivized to migrate to the minority chain to perform a 51% attack [172]. Several works have contemplated what happens when multiple cryptocurrencies share the same proof-of-work algorithm and some portion of miners automatically switch between networks in order to mine the more profitable coin [173, 174]. The primary finding of these works is that there exists a hash rate equilibrium between the competing networks based solely on the market price of their respective coins and that miners allocating their hash rate off-equilibrium merely creates a profitable arbitrage opportunity for other miners. As a result, miners "loyal" to the minority chain – that is, miners who will continue mining on the lower difficulty network – will mine alone if they exceed the equilibrium allocation. The security of the lower value network cannot be improved by allocating more hash rate below the equilibrium value, and loyal miners would centralize the network above the equilibrium value by causing other miners to migrate back to the stronger network. The only way to alter the equilibrium is to change the relative prices of the respective assets, which is non-trivial.

The relationship between a ledger's security and the price of its native asset appears to have better dynamics under certain conditions that are far more likely to hold when ASICs are involved in mining rather than just GPUs and CPUs. A combination of high fixed costs to buy hardware and set up a mining operation plus a low salvage value for the hardware itself leads to asymmetric hash rate changes in response to price changes [175, 176]. Specifically, when a network relies on ASICs, miners will deploy more hardware as prices increase but do not decrease to the same degree when prices decline. By responding less to adverse exchange rate shocks, ledgers secured by ASICs are less likely to suffer double-spend attacks after a price decline. Empirically speaking, Ethereum's hash rate responded to price symmetrically at first, but after ASICs were developed for Ethash (Ethereum's formerly ASIC-resistant proof-of-work algorithm), hash rate changes in response to price became more asymmetric [176]. This beneficial asymmetry is reduced if the ASIC is transferable to another cryptocurrency, though it is likely to still exist if the secondary network is small relative to the network they are actually mining on. A paper by Garratt and van Oordt explains the economic reasoning behind these insights [175]:

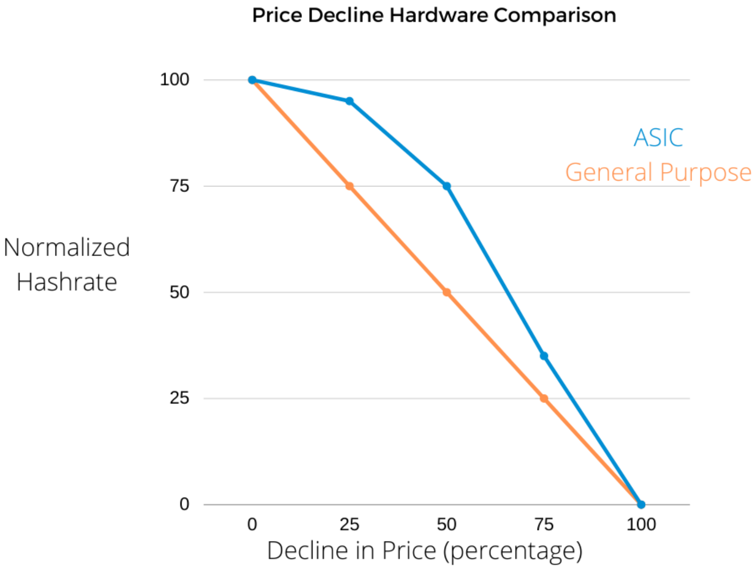


Fig. 21. In the face of an unanticipated price decline, the behavior of miners depends on the type of hardware they use. Because miners using general-purpose hardware can leave freely, they leave in proportion to the decline in the block reward. In contrast, miners who have already invested in ASICs, which cannot be used for other purposes, may continue to mine.

- Without fixed costs, the decline in the hash rate is proportional to the decline in the exchange rate due to free entry and exit into mining. With fixed costs and small exchange rate declines, miners continue mining to recover some of these fixed costs and will do so until it becomes more profitable to sell the hardware for scrap. ASICs, which are useless for anything other than mining on a particular ledger, have very low scrap value. This difference is shown in Figure 21.

- With fixed costs and low scrap value, launching a double-spending attack results in a loss in the present value of future mining rewards due to the attack's negative impact on the asset's price. Without fixed costs, the only costs of attempting a double-spending attack are the potential loss of mining rewards during the attack if it fails and the variable costs of mining for the duration of the attack.

- With fixed costs and low scrap value, the determination of whether a double-spending attack is profitable or not is path-dependent. At any exchange rate, the hash rate will be higher if the current exchange rate is less than a previous peak because mining operations will expand as the price approaches its peak. The price decline implies a decreased present value of future mining rewards, such that profitable double-spending attacks are more likely if the current price is the result of a decline from a previous peak. The principle here holds regardless of the hardware used but presents a more substantial security risk for general-purpose hardware due to the first point above.

- When multiple cryptocurrencies are mined with the same hardware and an attack on one of the networks is not expected to impact the price of other cryptocurrencies that can be mined with the same equipment, double-spending attacks are more likely to be profitable than if only a single cryptocurrency were available. Whether an attack can be expected to reduce the price of alternative cryptocurrencies using the same hardware likely depends on their relative sizes. It is hard to imagine a successful attack on the larger cryptocurrency not causing a decline in value for the smaller one because it demonstrates the insecurity of the smaller network. An attack on the smaller one is less likely to have the same impact and is unlikely to significantly affect the expected return from a given piece of hardware.

These arguments strongly suggest that there is a security advantage to a network when it is the dominant application of a given piece of hardware. However, the wider availability of general-purpose hardware offers other advantages that are difficult to measure but worth considering.

9.1.3. Mining Centralization in Practice

Some believe that – due to economies of scale – proof of work in the long term is likely to result in a concentration of power among an oligopoly consisting of a handful of large miners. For instance, in the model presented in [177], even relatively small economies of scale resulted in few miners operating simultaneously, and a miner with an $X\%$ cost advantage over any other miner will have at least $X\%$ of the total hash rate. As a result, a world in which a miner is able to gain a significant cost advantage over the competition is likely to result in a centralized mining network. That said, the model assumes that all potential miners decide whether to make investments in equipment simultaneously at particular factor prices. In the real world, these decisions are not made simultaneously, and prices change, likely disturbing the equilibrium analysis.

It is difficult to obtain data on the distribution of hash rate among miners because block reward payouts go to the pools rather than to the miners operating the hardware. That said, Romiti et al. performed an empirical analysis of Bitcoin mining shares and inferred that the addresses paid out from coinbase transactions likely belong to miners of the pool [178]. Their data came from three large mining pools and found that fewer than 20 distinct entities collect more than half of the rewards within each pool and that rewards from multiple pools often go to the same entity. This may sound like a substantial degree of mining concentration, but many of these payout addresses belong to exchanges, which dramatically complicates the analysis. Every distinct miner who directly sends their payouts to the same exchange will be considered a single entity under this methodology. As such, it is difficult to draw conclusions about how many actual miners operate on the network and their relative computational power.

Regardless of the concentration among miners, it is clear that the majority of miners will tend to form only a relatively small number of pools. As described above in Section 9.1.1, the centralization into relatively few pools does indeed present risks. For example, majority attacks have been conducted against many small and medium-sized networks where that ledger was not the dominant application for the relevant mining hardware. Further, even Bitcoin had a brief period in 2014 where a mining pool – GHash.io – had more than half the total hash rate of the network, though no attacks occurred at that time. On a more optimistic note, the ability of miners to allocate their hashes between multiple pools tends to lead to the decentralization of pools as larger pools increase their fees [179]. Large pools raise fees in order to decrease their hash rate because their positive impact on the difficulty level harms them more. In practice, distributing hash power this way is easy for miners. Tools exist to help and have been experimentally shown to increase a miner's Sharpe ratio (i.e., average excess reward over its standard deviation/volatility) by 260% compared to more passive miners [180]. Additionally, the adoption of Stratum v2 is likely to go a long way toward mitigating these pool concentration risks in the future.

In addition to miners and pools, hardware manufacturing and design is another potential avenue of centralization for proof-of-work cryptocurrencies. Designing ASICs requires specialized knowledge and capital, and there have historically been very few SHA-256 ASIC manufacturers (and even fewer for other proof-of-work algorithms). This market concentration extends to the foundries responsible for manufacturing the integrated circuits. That said, in the years since the first Bitcoin ASIC was produced in 2013, ASIC replacement cycles have grown substantially longer [181]. Early designs were obsolete within a few months, but designs have caught up with Moore's Law, and ASICs now often last years before more efficient designs make them unprofitable. This has opened up new competition in ASIC design, resulting in far more competition in the Bitcoin ASIC industry [181]. As it becomes harder and harder to squeeze out further optimizations for these chips, ASIC production is expected to become further commoditized, reducing the power of manufacturers.

Besides general concentrations of power, there may be other concerns regarding geographic centralization in certain regions or political jurisdictions. If miners or mining pools are concentrated in relatively few geographical areas, it becomes substantially easier for the governments of those jurisdictions to disrupt the network. It also creates certain risks that might not otherwise exist, such as flooding in a region popular for mining, which can take a large fraction of the network hash rate offline simultaneously.

There are a number of reasons why miners may cluster into relatively few geographical regions. First, miners are inclined to set up their facilities wherever electricity is cheapest, since it is the greatest variable cost for mining and thus one of the simplest ways to gain an advantage over the competition. In practice, a number of governments offer generous electricity subsidies, which tends to attract miners. Second, the incentives of mining pools are such that they benefit most by propagating their blocks to more than 50% of the hash rate as quickly as possible but not necessarily 100% for network latency-related reasons, as discussed in Section 10.2.1. This latter incentive is particularly relevant when governments or ISPs engage in network censorship that may slow down the distribution of blocks. If the majority of the hash rate is located in such a jurisdiction, then miners outside of that jurisdiction face a distinct mining disadvantage.

However, there are several factors that can mitigate the risks of geographic concentrations of miners. For one thing, there are significant operational challenges with seizing control over all of the deployed physical machines within a country. It is relatively easy to relocate machines, which is often done to take advantage of different electricity prices at different times of year. In addition, the hardware is already spread out across a variety of regions throughout the country. Seizing the vast majority of this equipment would be very challenging to do quickly and discreetly, and once word has gotten out that such seizures are happening, the community can plan for emergency actions, such as a hard fork to change the proof-of-work algorithms and render those machines useless. It would be far easier for the government to target the pools instead of the miners, but again, it would be challenging to do so covertly. Mining pools can quickly be set up in other locations around the world, and it is easy for miners to switch pools. Ultimately, when Stratum v2 is more widely used by these pools, the attack vector will be closed.

9.2. Difficulty Adjustment Algorithms

Since proof-of-work cryptocurrencies lack a central authority responsible for determining who can mine and with what machines, there will inevitably be a varying amount of hash power deployed to the network as the exchange rate and mining profitability changes. Nevertheless, in order to maintain a specific monetary policy and provide a better user experience, new blocks need to be found at a relatively consistent pace (e.g., roughly every 10 minutes in Bitcoin), regardless of the amount of hash power deployed. In these systems, the *difficulty adjustment algorithm* (DAA) is responsible for adjusting the difficulty of the mining puzzle in response to changes in the network's total hash rate in order to keep block production consistent. Without a DAA, an increasing hash rate would result in blocks being found more and more frequently, inflating the currency more quickly and making payments less predictable and secure for users.

Although the primary purpose of a DAA is to maintain consistent inter-block arrival times over the long term despite fluctuations in hash rate, there are a variety of possible considerations that may go into its design. The algorithm should ideally avoid sudden difficulty changes when the hash rate remains constant, discourage wild oscillations from the feedback between hash rate and difficulty, and avoid exceptionally long intervals between new blocks. Generally speaking, DAAs face a trade-off between being more stable or being more quickly responsive to changes in hash rate. More stable algorithms, such as Bitcoin's

DAA, result in more blocks that are "inappropriately" cheap or expensive because they measure hash rate less accurately and are more disruptive to users if the hash rate drops suddenly, but they are more secure and easier to reason about [182–184]. In Bitcoin, the difficulty is adjusted every 2,016 blocks (about two weeks) proportional to the degree to which blocks were found too quickly or slowly. For instance, if it takes four weeks to mine 2,016 blocks, the difficulty for the following 2,016 block epoch will be cut in half. Algorithms that adjust the difficulty more frequently will maintain more consistent block times but are easier for miners to game for unfair advantage [185]. Common DAAs include variants of simple moving averages and exponential moving averages that adjust after every block, though there is a wide design space.

In a world where more than one proof-of-work ledger exists, there is a risk that difficulty changes in one network cause a behavioral shift among miners that affects another network. For example, an interaction between the DAAs of Bitcoin and Bitcoin Cash (which share the same hardware) caused wild fluctuations and instability in the Bitcoin Cash hash rate until a hard fork changed the Bitcoin Cash DAA [186]. More generally, *coin-hopping attacks* are possible between networks that share the same hardware [187]. At the beginning of a difficulty epoch, an adversarial or strategic miner can switch to mining on an alternative network and then switch back to the original network at the beginning of the following difficulty epoch when the difficulty is lower due to the adversary's withdrawn hash rate. This allows the miner to mine at a lower average difficulty and cost than competing miners that do not hop between coins. In some circumstances, it may be profitable to perform similar strategic deviations for a single network, mining during one difficulty epoch and turning machines off during the next, or using only a portion of the miner's available hash rate during some epochs in order to lower the difficulty for future epochs [188, 189].

9.3. Attacks Against Mining Pools: Pool-Hopping and Block Withholding

Mining pools allow miners to lower the variance in their revenue by allowing them to split the block reward based on how many shares each miner submits to the pool operator. An intuitive scheme for doling out a pool's rewards to its constituent miners would be to define a round as the period between two blocks being found by a given pool and then have individual miner payouts be proportional to the number of shares that each miner submitted during the round. Unfortunately, while this proportional reward allocation may be simple, it is also insecure due to the *pool-hopping attack* [190]. In fact, any pool reward-sharing scheme where the profitability of mining depends on the current state of the pool is potentially subject to pool-hopping.

In a pool-hopping attack, a miner will join a pool, mine within it for some period of time, and then leave for another pool in such a way that they receive a disproportionate amount of revenue at the expense of other miners who remained in the pool. Under the proportional reward scheme, the payout for every share is equal to the block reward divided by the number of shares submitted in the round. As more time passes within a round, more shares are submitted, and thus the reward per share decreases. Therefore, an individual miner benefits if they submit shares to the pool during its shorter rounds but mine in other pools during longer rounds. This can be approximated simply by mining within a given pool for a short period and then switching to another pool if the round has not completed yet. As a result of this attack, honest miners who continuously participate in the pool can, in the worst case, receive 43% less than their "fair share." Unsurprisingly, mining pools no longer use the proportional payout scheme.

Another attack that can be launched against mining pools is the *block withholding attack*, where a malicious miner submits valid shares to the pool operator but does not submit blocks when a full proof of work is found [190]. Block withholding can be especially nasty when mining pool operators are also miners. With multiple competing mining pools, the Nash equilibrium is a mixed strategy where miners use some of their hash rate to infiltrate competing pools to withhold blocks [191, 192]. This is analogous to the famous prisoner's dilemma in game theory: while everyone would be best off if all parties refrained from block withholding, each party benefits from their own malicious deviation. The result is that all parties will deviate and withhold blocks, and all parties will be worse off. If mining pools are attacking each other in this way, the total potential network hash rate will not be fully utilized, and the network will be less secure than it otherwise would be. If the network hash rate is static during a difficulty epoch, the attack only becomes profitable after the next difficulty adjustment. Until then, block withholding lowers the revenue for both the attacker and the victims, though it harms the victims to a greater degree. This has occurred in the real world, such as when the Eligius mining pool suffered a 300 bitcoin loss in June 2014 due to block withholding [193]. However, the need to wait for a difficulty epoch before it becomes profitable and the low returns from the attack combine to make it relatively uncommon. A related but much stronger attack – *fork after withholding* – operates the same as block withholding, except that when the attacker detects that a competing block from a different pool is found, they release the withheld block in order to create a deliberate fork [194].

It is possible to mitigate the threat of block withholding through an adjustment to the pool payout scheme. For example, the "incentive compatible" scheme described in [195] makes block withholding unprofitable. Let D be the expected number of shares found per full proof-of-work solution (i.e., D is the ratio between the full difficulty and the share difficulty). When a miner within a pool finds a block and submits it to the pool operator, D is compared with the number of shares submitted in that round, S. If $S \geq D$, then rewards are proportional. On the other hand, if $S < D$, each share receives a fixed reward of $\frac{1}{D}$, while the remainder of the available block reward goes to the miner who submitted the full solution. By providing a larger reward for the miner who found the block, they are incentivized to report it. Unfortunately, this reward scheme is also susceptible to pool-hopping, as with the proportional scheme, and is therefore not appropriate for real-world use. Most existing mining pools follow one of three styles of alternative reward-sharing mechanisms:

1. *Pay-per-share (PPS)*: Each submitted share immediately credits the miner with a fixed payout, and the pool operator keeps the full block reward when blocks are found. This minimizes variance-related risks for the miner [196] but turns the pool operator into a financier who takes on the full mining variance risk. During short rounds, the pool operator wins, whereas they lose money during long rounds in which many shares are submitted. Due to this risk, PPS pools tend to charge miners higher fees. PPS is fully immune to pool-hopping attacks.

2. *Pay-per-last-N-shares (PPLNS)*: This method eschews the idea of rounds, and instead, the pool operator maintains a queue of the most recent N shares submitted to it. Whenever a miner within the pool finds a block, payouts are proportional to the number of shares each miner found that remains on the queue. A randomized version of PPLNS is also possible, where instead of a queue in which the oldest share is removed when each new share is found, a random one is removed [197]. While mining reward variance is higher for PPLNS than PPS, it has the lowest variance for schemes where the mining pool does not risk running out of funds and having a deficit with respect to the miners [196].

   This payout scheme is less susceptible to pool-hopping than the proportional method but is not hopping-proof. Miners can benefit from joining a pool immediately before the difficulty is about to decrease and leave a given pool when the difficulty is about to increase [190]. PPLNS is fairly robust, but there are situations where miners can benefit from strategic deviations, such as hoarding a certain number of shares and only submitting them to the pool upon finding a block. This can prevent old shares from exiting the queue [198].

3. *Share-scoring methods*: This includes the original Slush method and the superior geometric method that it inspired, which is fully resistant to pool-hopping. These schemes give each newly submitted share a score that depends on how much time has elapsed since the beginning of the current round: as more time passes, the score increases. At the end of the round, payouts are proportional to the total score rather than the total number of shares. Early shares are worth more than late shares when using the proportional method, but the increasing scores over time help to counter this effect by reducing the value of early shares. The geometric method includes a variable fee that is higher earlier in the round but decays throughout and is parameterized such that the score given to any new share is always the same relative to both existing and future scores. This removes any advantage that a miner could gain by timing when to enter or leave the pool [190].

Note that the victim of a block withholding attack depends on the payout scheme used. If PPS is used, then the pool operator is the victim because they do not get to collect the block reward yet must still pay a constant amount to each miner for their shares. In methods that do not have operator risk, such as proportional, PPLNS, and geometric, the other miners within the same pool suffer when block withholding attacks are performed.

```
on Init
    public chain ← publicly known blocks
    private chain ← publicly known blocks
    $privateBranchLen \leftarrow 0$
    Mine at the head of the private chain.

on My pool found a block
    $\Delta_{prev} \leftarrow$ length(private chain) − length(public chain)
    append new block to private chain
    $privateBranchLen \leftarrow privateBranchLen + 1$
    if $\Delta_{prev} = 0$ and $privateBranchLen = 2$ then      (Was tie with branch of 1)
        publish all of the private chain                    (Pool wins due to the lead of 1)
        $privateBranchLen \leftarrow 0$
    Mine at the new head of the private chain.

on Others found a block
    $\Delta_{prev} \leftarrow$ length(private chain) − length(public chain)
    append new block to public chain
    if $\Delta_{prev} = 0$ then
        private chain ← public chain                        (they win)
        $privateBranchLen \leftarrow 0$
    else if $\Delta_{prev} = 1$ then
        publish last block of the private chain             (Now same length. Try our luck)
    else if $\Delta_{prev} = 2$ then
        publish all of the private chain                    (Pool wins due to the lead of 1)
        $privateBranchLen \leftarrow 0$
    else                                                    ($\Delta_{prev} > 2$)
        publish first unpublished block in private block.
    Mine at the head of the private chain.
```

(a) Bitcoin DAA

```
DEFINITIONS:
$effectiveState := sumWork(privateChain) - sumWork(mainChain)$
$ifLose := effectiveState - nextMainChainBlockDifficulty$
INIT:
$effectiveState \leftarrow 0$
$ifLose \leftarrow -(nextMainChainBlockDifficulty)$
ON SELFISH MINER FINDS BLOCK:
append new block to private chain and continue mining on private chain
$effectiveState \leftarrow effectiveState + newPrivateBlockDifficulty$
$ifLose \leftarrow ifLose + newPrivateBlockDifficulty$
ON OTHER MINERS FIND BLOCK:
append new block to public, main chain
$effectiveState \leftarrow effectiveState - newPublicBlockDifficulty$
$ifLose \leftarrow ifLose - newPublicBlockDifficulty$
if $ifLose \leq 0$ and $len(privatechain) > 0$ and $effectiveState > 0$ then
  publish private chain, overtake main chain, and mine on top of new public chain tip
else if $effectiveState = 0$ and $len(privatechain) > 0$ then
  publish private chain and enter race
else if $ifLose > 0$ then
  continue mining on private chain
else if $len(privatechain) = 0$ then
  mine on top of new public chain tip
end if
```

(b) Variable DAA

Fig. 22. Selfish mining strategy. With a DAA that adjusts the difficulty every block, the $effectiveState$ variable captures the difference in total work between the selfish miner's private chain and the main chain. For Bitcoin's DAA, it is sufficient to keep track of the difference in blocks instead of work. The $ifLose$ variable represents what the difference in work would be if the honest miners won the next block. It is used to address situations in which the selfish miner currently has the most work chain but would give up if honest miners win the next block. [185, 199]

9.4. Selfish Mining

Satoshi Nakamoto believed that in the Bitcoin system, it is in miners' rational interest to behave honestly so long as the adversary does not control more than half of the network's computational power [1]. This was later shown to be false due to the *selfish mining* phenomenon, where strategically withholding blocks from the network can be more profitable than honest mining, even with only a minority of the network's hash rate [199–201]. While the original attack was described assuming constant difficulty blocks, the strategy was expanded to account for variable difficulty blocks in [185]. The attack is relevant to Nakamoto Consensus (see Section 10) but also applies to longest-chain proof of stake and several other proof of work protocols.

Selfish mining works by forcing honest miners to waste their energy mining blocks on a chain that is "destined" to be reorganized and become stale. When the attacker successfully mines blocks, they keep them private and only reveal them to the rest of the network when the honest chain is closer to being caught up with the adversarial chain. The attack allows the selfish miner to gain a disproportionate share of the mining rewards (i.e., it reduces chain quality and improves the adversary's *relative revenue*) and becomes profitable sometime after the difficulty level drops to reflect the reduced block production rate. The detailed strategy is listed in Figure 22a for the case of Bitcoin's DAA (or any DAA where the difficulty remains constant for an epoch before changing) and in Figure 22b when the difficulty changes with every new block.

The attack is parameterized by two variables, α and γ, with $0 \leq \alpha, \gamma \leq 1$. α is the fraction of the total hash rate that is engaged in selfish mining, and γ represents the fraction of honest miners that will add blocks to the chain on top of the selfish miner's block during a block race (and thus relates to the adversary's control over the network). γ can be considered a measure of the rushing capabilities of the adversary in that it captures the advantage that the selfish miner has when reacting to blocks found by honest peers.

Whether the selfish mining strategy is rational or not depends on the values of α and γ. The hash rate where a selfish miner can improve their relative revenue above and beyond what they would earn from mining honestly is given in the following inequalities:

$$\frac{1-\gamma}{3-2\gamma} < \alpha < \frac{1}{2} \tag{1}$$

If $\gamma = \frac{1}{2}$, selfish mining is profitable when $\alpha \geq \frac{1}{4}$, and if $\gamma = 0$, selfish mining is profitable when $\alpha \geq \frac{1}{3}$. The $\gamma = \frac{1}{2}$ scenario corresponds to the situation in which miners – upon seeing two equal-work chain tips – choose randomly between them. Further, if $\gamma = 1$, selfish mining is beneficial to the attacker at any hash rate. In the optimized attack from [202], the required adversarial hash rate is decreased from 25% to 23.21% when $\gamma = \frac{1}{2}$.

Nayak et al. proposed three additional "stubborn" variants of the strategy, which may perform better than selfish mining under some conditions [203]:

1. *Lead stubborn*: When a selfish miner has a lead of two blocks, but the honest miners find a block and cut the lead to one, the selfish miner publicizes their longer chain to win the race. In contrast, a lead-stubborn miner will only publicize the first block to "match" the honest chain and try to cause a race.

2. *Equal-fork stubborn*: In normal selfish mining, when there is a block race between equal length forks and the attacker mines a block, they reveal it immediately to win the fork. In equal-fork stubborn mining, they would withhold the new block and just have a one block lead again.

3. *Trail stubborn (T_j-stubborn)*: When a selfish miner falls behind the honest chain, they simply adopt the honest chain. A T_j-stubborn miner will continue mining on their private chain until they are $j+1$ blocks behind the honest chain.

Selfish mining is one of the most well-studied attacks in the permissionless ledger literature and has been investigated under a wide variety of scenarios. For example, heterogeneous network connectivity among honest miners helps a selfish mining attacker because γ rapidly increases as the variance in block propagation delay increases [204]. In [205], selfish mining was found to be far more profitable with larger network delays, but technologies that speed up block propagation – such as compact blocks and relay networks – effectively eliminate this difference for blocks of moderate size (compact blocks and relay networks are discussed in Section 10.2.1).

Most selfish mining research assumes the existence of a single adversary, but a line of work investigates what happens with multiple selfish miners simultaneously operating on the same network [206–209]. Unsurprisingly, the security of the network is further reduced with multiple selfish miners. For example, with two independent selfish miners, the threshold for profitably selfish mining drops from 25% to 21.48% [206]. As the number of simultaneous selfish miners increases, the profitability threshold for each miner decreases [207, 209]. A necessary condition for n selfish miners to simultaneously benefit from the attack is that each of their individual hash rates α_i is in the range $\frac{1}{n+1} < \alpha_i < \frac{1}{n-2}$, such that up to seven adversaries can operate with 12% of the hash rate each [209].

The profitability of selfish mining depends on the network against which it is applied. The DAA is a significant factor. For example, in systems where the difficulty is able to rapidly adjust downward when blocks are slow, selfish mining tends to be more profitable. When the DAA only looks at a relatively short period of time as the basis for the adjustment, selfish miners can dramatically increase their profits by manipulating the timestamps they include in block headers [185].

Ethereum, in particular, has received the most attention outside of Bitcoin [210–212]. Ethereum's consensus algorithm, GHOST (discussed in Section 11.2), differs from Bitcoin's in that it includes the existence of stale blocks ("uncles") as part of the DAA as well as the reward scheme. Most significantly, Ethereum miners are rewarded even if their blocks do not make it into the main chain so long as they are referenced as uncle blocks shortly after being mined. As a result, the threshold for selfish mining profitability is reduced because even when the attacker loses a block race, they are likely to still get some reward for taking the risk. Using the observed uncle block ratio from Ethereum in December 2017, [211] showed that the profitability threshold for selfish mining was $\alpha = 0.185 \pm 0.012$. Another model found that selfish mining was profitable with $\alpha > 0.163$, and that beneath this value, the selfish miner loses far less than they would by attacking Bitcoin [212].

Despite being perhaps the most significant theoretical attack on many permissionless systems (aside from the majority attack), significant selfish mining attacks have not been observed in the real world. There are a number of reasons why this may be the case:

- Executing the selfish mining strategy requires specific software for implementation. The expertise to develop this is not widespread and the software itself is likely to be bug-prone and challenging to test. Due to the costs imposed by proof of work, a bug could cause tens or hundreds of thousands of dollars in losses for the attacker.
- Most models assume that the exchange rate for the asset being mined selfishly remains constant throughout the duration of the attack and beyond. However, if selfish mining was discovered on a network, one might reasonably expect it to decrease the exchange rate of the asset in question, which can dramatically reduce the profitability of the attack (especially if the relevant hardware cannot be repurposed).

- For larger networks, like Bitcoin, it would be prohibitively expensive for individual miners to acquire enough computational power to profitably perform the attack. While mining pools can perform selfish mining, it is likely to be discovered quickly, at which point the individual miners can defect to honest pools. Unfortunately, these miners are incentivized to remain in the pool and take advantage of the profits that come from selfish mining so long as the exchange rate remains constant (but this assumption is questionable). Selfish mining detection is facilitated by the common behaviors of individual miners belonging to multiple pools, as well as pools monitoring each other.
- If enough computational power can be acquired to perform selfish mining, it is often easier and perhaps more profitable to double-spend with a majority attack. This is likely the case for small networks where markets exist to rent significant portions of the hash rate.
- The parameter γ is important to the profitability of the strategy, but estimating it is challenging, and it is likely to change throughout the duration of the attack.
- Selfish mining is not profitable until sometime after the difficulty adjusts downward to reflect the adversary "dropping off" the network. Most research assumes that no new miners are enticed to (honestly) mine on the network based on this lower difficulty, which may not be realistic. If this assumption turns out to be false, the selfish miner may never recoup their losses from prior to the difficulty adjustment.

On the other hand, selfish mining immediately lowers the profitability of honest miners. If this induces some of them to quit mining, selfish mining becomes even more profitable for the attacker [213].

10. Nakamoto Consensus

Satoshi Nakamoto first proposed what has been dubbed "Nakamoto Consensus" in 2008, and it ushered in a new era of consensus algorithm research and innovation [1]. The term "Nakamoto Consensus" is used frequently and inconsistently in the consensus literature. Sometimes, it is used merely to describe the longest chain rule without specifying what mechanism is used to prevent Sybil attacks. However, that is not the definition used here. In this document, Nakamoto Consensus is defined by two things:

1. Proof of work is used as the Sybil-resistance mechanism and for randomized leader election.
2. The longest chain rule (LCR) is used as the fork-choice rule to form a blockchain. Technically, the LCR is a misnomer because the "longest" chain is defined as the heaviest one or the one that proves the most work rather than the chain with the greatest number of blocks.

Nakamoto Consensus is most prominently used in Bitcoin but is also one of the more commonly deployed consensus models overall. In the permissionless design space, it is by far the most well-understood protocol with formal security proofs under a wide variety of models. Having been deployed in Bitcoin since January 2009, Nakamoto consensus has the most real-world battle testing of any permissionless protocol. The assumptions upon which its security is derived are simpler than most:

1. **Honest majority.** The majority of the work that can be applied to the network is being used by honest participants rather than malicious ones. See Section 10.2.2 for more details.
2. **Bounded network delay.** Messages that are sent by participants must propagate to the rest of the network within some bounded period of time. See Section 10.2.1 for more details.
3. **Collision-resistant hash function.** A collision-resistant cryptographic hashing function is required in order to maintain the integrity of the blockchain itself and ensure that each block acts as a commitment or vote on all preceding blocks in the chain.

Nakamoto Consensus is also very efficient in terms of minimizing communication overhead. Other protocols often require the exchange of additional information regarding which validators have already seen which blocks or allow duplicate or invalid transactions inside of blocks that are only later filtered out. In addition, the simple blockchain structure – as opposed to multiple parallel chains or DAGs – ensures that transactions are globally ordered as soon as blocks are generated. This reduces transaction confirmation latency and is compatible with all smart contract programming models. In contrast, using a different structure requires some care to ensure that the transactions are totally ordered so as to be suitable for smart contracts. As such, it would not make sense to describe other, more recent consensus algorithms as strict improvements over Nakamoto Consensus but rather as having trade-offs. Proof-of-work variants that stay close to Nakamoto Consensus are described in Section 11.1, while a wider variety are discussed throughout Section 11.

The mechanics of Nakamoto Consensus are simple. All nodes begin with the *genesis block*, a common reference string that initializes the state of the system. Miners then search for proof-of-work solutions to a random puzzle that includes the genesis block as an input (as well as a list of transactions and some other metadata). A miner who finds a proof-of-work solution is elected leader and produces a block, which is then published and gossiped over a peer-to-peer network. When other nodes see this block, they validate the proof of work and the transactions, forward the block to their peers, and begin trying to find solutions to the next puzzle using the new block as input. Over time, this builds a chain of blocks that extend from the genesis block, adding transactions to the ledger and updating the system state.

Due to random chance, network delays, or malice, it is possible for more than one competing proof of work to be found at the same height in the blockchain. For instance, if

Alice and Bob both find blocks at height one immediately after the genesis block, then it is unclear whether miners should consider legitimate and thus build new blocks on top of Alice's chain or Bob's chain. In this case, there is a tie, and miners can choose arbitrarily (though they typically prefer the one they saw first). At some point, a miner will extend the chain from either Alice's block or Bob's block, and then that chain will have the most work backing it. This breaks the tie, and all honest nodes will switch to the chain with the new block because honest nodes follow the chain with the most work supporting it. This process continues indefinitely with honest miners building blocks that extend the longest (most work) chain they are aware of and results in a growing ledger of transactions.

10.1. Theory of Nakamoto Consensus

Nakamoto Consensus (and sometimes just Bitcoin) is frequently brought up as a solution to the Byzantine Generals Problem (see Section 1.1), thus making it a broadcast algorithm. At other times, it is presented as an agreement algorithm. Despite these frequent contentions, it is not strictly true that Bitcoin and Nakamoto Consensus perfectly fit into these paradigms.

In particular, the "validity" property of BGP and BA does not hold for Nakamoto Consensus. Recall that BGP validity guarantees that if the leader is honest and begins with input value v, then all honest nodes decide v. BA validity guarantees that if all honest nodes propose the same value, then any correct node must decide v. There is also weak validity, which requires that each honest node's output must be the input of some honest node. The randomized proof-of-work process interferes with achieving these notions of validity. Unless the adversary has a negligible fraction of the total computing power of the network, they will eventually be first to produce a proof of work, at which point all honest nodes will immediately switch to and extend the chain from this adversarial block and abandon the original "honest" input.

Similarly, the probabilistic guarantees provided by Nakamoto Consensus as a result of its proof-of-work process can be framed as providing agreement but not termination or, alternatively, termination but not agreement [214]. Ultimately, Nakamoto Consensus satisfies "eventual consistency" or "probabilistic consistency," where the probability of agreement on a particular block at a particular position increases exponentially as additional blocks extend the chain's tip from the block in question. One can, therefore, consider Nakamoto Consensus to satisfy agreement but not termination: if the blockchain protocol executes forever, then agreement will hold probabilistically with probability 1. Alternatively, Nakamoto Consensus may satisfy termination but not agreement: if x blocks exist and the system runs for a finite duration of k blocks, then consensus is achieved for block x, but the probability that agreement is satisfied is less than 1 and grows exponentially with k.

By now, there is a considerable amount of literature proving that Nakamoto Consensus and Bitcoin provide particular security properties under a variety of assumptions [9, 165, 182–184, 215–230]. Ultimately, Nakamoto Consensus creates a state machine that operates over a distributed ledger that satisfies *persistence* and *liveness* with security parameter k, sometimes called the "Bitcoin Backbone."

- **Persistence.** Once a transaction is at least k blocks "deep" in the blockchain of an honest node, then with overwhelming probability, the same transaction will be included at the same position of every other honest node's blockchain and remain there permanently.
- **Liveness.** Transactions sent by honest clients will eventually be at a depth of more than k blocks in honest nodes' blockchains.

The first security proof for Nakamoto Consensus in synchronous networks is from [9], where persistence and liveness follow from the *common prefix* and *chain quality* properties. It was then shown in [215] that an additional *chain growth* property was, in fact, required for liveness.

- **Common prefix.** For a security parameter k, the blockchains of two honest nodes differ only in the last k blocks from the tip. That is, if one were to "chop off" the final k blocks of two honest nodes' chains, one of the resulting subchains would be a prefix of the other. A related requirement is *future self-consistency*, such that at any two points in time, with high probability in the security parameter T, the chain of an honest node differs only within the last T blocks (i.e., alternating between two completely different chains is not allowed).
- **Chain quality.** For parameters μ and T, it holds that for any T consecutive blocks in the blockchain of an honest party, the ratio of honest blocks is at least μ. Stated differently, it should be the case that "enough" of the blocks that end up in the blockchain were proposed by honest nodes.
- **Chain growth.** For parameters τ and s and any honest party P with chain C in a given view, it holds that for any s rounds, there are at least $\tau * s$ blocks added to the chain of P. In other words, the blockchains of honest nodes must continuously grow at a certain pace over time.

The model used in [9] assumed a fixed set of n equally powerful miners (but with n unknown), where each party can make the same number of queries, q, to a hash function modeled as a random oracle. This is considered a "flat" model because each miner is equal, but one can imagine differences as being represented by real-world entities controlling variable numbers of players. The adaptive, rushing adversary controls $f < \frac{n}{2}$ of those parties and can thus make $f * q$ proof-of-work queries per round. In this environment, if $\frac{f}{n-f} < 1$, honest parties will have blockchains with large common prefixes when pruning the most recent k blocks from the tip of the chain, except for some small probability that drops exponentially with k. Further, the blockchains of honest parties will include blocks from both honest and malicious parties, but so long as the majority of the hash power is honest, the ratio of blocks produced by honest parties compared to malicious parties is bounded by

$\frac{f}{n-f}$. This chain quality is fairly low unless there is a large majority of honest miners (see Section 9.4 on selfish mining for one adversarial strategy that reduces chain quality). For example, if an adversary controls $\frac{1}{3}$ of the hash power, they may end up contributing nearly half the blocks. Finally, if the network is unable to remain sufficiently synchronized (i.e., if the message latency between honest miners approaches the expected time that it takes to find proof-of-work solutions), then maintaining the common prefix property requires a larger and larger hash rate (super)majority.

The analysis in [216] provides security proofs in the "Δ-bounded delay" model. This model comports with the standard definition of partial synchrony presented in [26], where there exists a fixed, unknown upper bound Δ on message delay. However, it does not fit the game-based description that immediately followed in [26], where the protocol designer designs a protocol and the adversary then chooses Δ. Interestingly, this makes systems that depend on Nakamoto Consensus, like Bitcoin, behave like asynchronous or partially synchronous protocols while still technically being synchronous.

It is instructive to consider why it is more challenging to address network delays in a proof-of-work model than in most permissioned systems. To transform a synchronous permissioned protocol into a Δ-bounded delay one, honest replicas can simply wait Δ time steps before responding to any messages. In the permissionless setting using proofs of work, the adversary can increase its use of computational resources by a factor of Δ to try to solve puzzles while the honest miners "wait."

In the context of Nakamoto Consensus, the delay must not only be bounded but sufficiently bounded. An arbitrary but finite delay will not suffice. If the delay were truly arbitrary (as it would be in an asynchronous network), an adversary with a small fraction of the hash rate could simply delay the arrival of honest parties' messages long enough to guarantee that the adversary creates an even longer chain than the ones acknowledged by honest nodes, forcing honest nodes to adopt the adversarial chain. Without any adversarial hash power, it is still possible for this network adversary to cause common prefix violations. However, in a model where the adversary can delay messages with only some probability less than 1, Nakamoto Consensus can remain secure even when delays may be quite a bit larger [217], which may motivate the use of satellite or radio technologies to broadcast blocks in ways that are more difficult for adversaries to disrupt.

Even under partial synchrony, if ρ is the fraction of the hash power controlled by the adversary, n is the number of miners, and p is a mining hardness parameter that captures the probability that a single random oracle hash query generates a valid proof of work, the common prefix property (and thus persistence) will not hold when $p > \frac{1}{n\rho\Delta}$.

As a result, the mining puzzle's difficulty must be appropriately set as a function of the maximum network delay in order to maintain security. The primary result from [216] is that so long as $\rho < \frac{1}{2}$, then for any delay bound Δ, there exists a sufficiently small p such that Nakamoto Consensus maintains consistency. The consistency proof relies on a concept dubbed *convergence opportunities*, which provide honest nodes the opportunity to synchronize themselves and converge upon the same blockchain. A convergence opportunity occurs when:

1. There is a period of Δ rounds where no honest miners mine a block.
2. A round occurs where a single honest miner mines a block.
3. Another period of Δ rounds occur without any honest miners mining a block.

Note that in step 2, only a single honest miner can mine a block during that time. If multiple honest miners mine competing blocks, there is no convergence opportunity because honest miners may be split over which block they ought to mine on top of (later works tightened the security proof by relaxing this requirement [225, 226]). To prove the security of Nakamoto Consensus, [216] shows that convergence opportunities occur with sufficient frequency and that attackers can only prevent them by mining a block that is accepted by the honest nodes during this time. An adversarial block that prevents a convergence opportunity cannot have been mined and withheld particularly long before the time period in question because the honest mining majority will quickly produce and distribute a longer chain, making the adversarial block irrelevant. When the adversarial hash rate is sufficiently bounded, the number of adversarial blocks will be outnumbered by the frequency of convergence opportunities, guaranteeing that honest nodes will eventually converge on a consistent chain.

10.1.1. Nakamoto Consensus With Chains of Variable Difficulty

The above analyses assumed that the mining difficulty remained constant throughout the execution of the protocol. In actuality, the difficulty adjusts with the addition or removal of computing power so as to maintain an average target block interval (for instance, in Bitcoin, the target block interval is 10 minutes or 600 seconds). Several works extend the prior analyses to distributed ledgers where the difficulty changes over time [182–184].

In [182], the authors extend the synchronous model from [9] to account for Bitcoin's difficulty adjustment algorithm, which recalibrates the difficulty parameter after every 2,016 blocks (approximately two weeks) in order to account for changes in hash rate during that time. The new model does away with the fixed number of miners n and instead allows the number of miners to vary. There are bounds on how quickly the number of players can change over time while maintaining consistency, and if these bounds are respected and miners do not deviate too far from expectation, the block production rate should remain sufficiently stable to maintain the common prefix and chain quality properties. The most crucial result from the analysis is that the length of a difficulty epoch needs to be sufficiently large in order to bound the probability of attacks, which helps to justify the lengthy epochs used in Bitcoin (while raising questions as to the security of more rapidly adjusting DAAs). Further, Bitcoin's DAA utilizes "dampening filters" such that the difficulty adjusts by no more than a factor of four in either direction, and this dampening is required in order to prevent attacks against the common prefix property [200].

Some limitations from [182] are addressed in other papers that account for variable difficulty [183, 184]. In particular, [182] relies on two key assumptions: 1) a synchronous network prevents the adversary from delaying network messages between honest parties, and 2) the adversary cannot adaptively choose how many players – and thus how much hash rate – is deployed in any given round but must instead schedule this in advance. Clearly, these assumptions do not hold in the real world, but [183, 184] prove the security of Nakamoto Consensus with an adaptive work schedule and Δ-bounded message delays. Note that changes in computing power must not occur too quickly or else the mining difficulty will not be appropriately set relative to the network delay. The only other condition required in the proof from [183] is that the initial difficulty is set appropriately to the network delay and available computational power. That is, if the difficulty is properly calibrated from the start and does not change too quickly, Nakamoto Consensus remains secure.

The difficulty adjustment algorithm employed in [183] differs slightly from that of Bitcoin. In particular, for an additional security parameter κ_0, the difficulty calculation "chops off" the first and last κ_0 blocks of the epoch, which helps maintain consistency and avoids the need to deal with epoch boundaries in the analysis. In addition, the timestamp validity rules for a given block differ slightly from that of Bitcoin: all timestamps must strictly increase, and honest nodes reject blocks where the timestamp is in the future. Assuming that the chain quality property holds, these rules ensure that an honestly produced block occurs periodically, forcing adversarial blocks to be located between two nearby honest blocks and thus preventing adversarial blocks' timestamps from deviating too far from the actual time. The analysis in [184] uses the real Bitcoin DAA but with similarly modified timestamp rules.

10.1.2. Additional Analyses of Nakamoto Consensus

There are quite a few additional security analyses of Nakamoto Consensus and Bitcoin that often use different models. For example,

- Bitcoin was proven secure with *universal composability* (UC) in [218] in networks with bounded delays. A UC-model proof is important because it means that a Bitcoin-like ledger can be arbitrarily composed with other protocols that may rely on it, such as running multi-party computations on top of a blockchain.
- The Bitcoin Backbone was proven secure in the quantum random oracle model in [220, 231], assuming a sufficiently bounded quantum adversary.
- In [221], the Bitcoin Backbone was proven secure when different miners have different hash rates (i.e., it avoids the "flat" model where each miner is considered to have an equal amount of hash power).
- The Bitcoin Backbone remains secure even when some of the nodes "prune" their blockchains by removing old blocks from their local ledger [224].

- A simple proof of the security of Nakamoto Consensus in the continuous time model (instead of using discrete rounds) was provided in [229].

- The trade-off between confirmation latency and security was formalized in [232], which provides guidance for optimizing throughput by choosing appropriate parameters for block size and expected block times.

- Permissionless longest-chain protocols, including Nakamoto Consensus, were proven secure in a generalization of the synchronous communication model, where message delays are independent, identically distributed, and thus potentially unbounded [233].

- Some works have attempted to remove the random oracle assumption from their security proofs. For example, [223] formalizes a primitive called "signatures of work," which enables honest majority consensus in permissionless settings analogously to how digital signatures allow honest majority consensus in permissioned settings. The "key" for these "signatures" is the most recent block hash seen by a miner, so the key ensures timeliness of the work. Unfortunately, the only known instantiations of the primitive rely on random oracles themselves. A standard model proof of security without the use of a random oracle for the Bitcoin Backbone protocol was provided in [165].

- Several works have tightened the consistency bounds for longest-chain protocols like Nakamoto Consensus [222, 225, 226, 230]. One reason why earlier bounds were not tight was due to the negative influence on security that occurs when multiple honest miners mine blocks close to each other (see the discussion on convergence opportunities in Section 10.1), and [226] uses proof techniques that mitigate the impact of this. The resulting consistency bound – where r_a is the expected number of adversarial proofs of work generated per unit time, r_h is the expected number of honest proofs of work, and the maximum message delay is Δ_0 – is

$$r_a < \frac{1}{\Delta_0 + \frac{1}{r_h}} \tag{2}$$

  The analysis in [222] produced security bounds in terms of the expected number of network delays until a block is mined, c (e.g., if it takes 10 seconds before all miners receive and validate a block and the expected block interval is 600 seconds, $c = 60$). In this case, if h is the honest fraction of the hash rate and f is the adversarial fraction, consistency is maintained so long as

$$c > \frac{2h}{ln(\frac{h}{f})} \tag{3}$$

- A particularly interesting result is that the Bitcoin Backbone can maintain its primary security properties (i.e., chain quality, chain growth, and common prefix) even in situations where a dishonest majority temporarily exists so long as there is an honest majority in expectation over time [227, 228]. This would capture situations where, for example, a mining pool has suffered a denial of service that creates a temporary dishonest majority of miners that will be rectified when the pool comes back online.

One more analysis that deserves special attention is [219], which uses the game-theoretic rational protocol design framework to show that – assuming a particular class of incentives within the Bitcoin protocol – the honest majority assumption can be replaced by the weaker assumption that miners seek to maximize their profits. The utility function that accounts for these incentives incorporates the block subsidy, transaction fees, and mining costs but does not include any incentives external to the protocol, such as exploiting the transaction ordering for front-running or shorting the bitcoin asset in order to profit from a decrease in its price. A number of important conclusions can be drawn from this:

- In the less realistic model where transaction fees are excluded from the analysis, Bitcoin is incentive-compatible, and even large coalitions of miners lack an incentive to deviate from the protocol so long as the price of Bitcoin is high enough for mining to be profitable.
- When transaction fees are included, the system remains incentive-compatible so long as fees are sufficiently large. However, the same is not true for arbitrary transaction fee distributions. If some fees are much higher than others, an attacker may try to build two chains in parallel with the high fee transactions and maintain the fork long enough to spend those fees on both chains. On the other hand, if an honest majority of computing power exists, the only profitable deviation by the adversary is to withhold high-fee transactions from the rest of the miners in order to claim them without the risk of a competitor collecting those fees. As a result, the security assumption for Bitcoin can be relaxed from needing an honest majority to needing many high fee transactions and only requiring an honest majority when fees are low.
- As the block subsidy in Bitcoin is reduced over time and miners increasingly rely on transaction fees for revenue, there must be a significant transaction fee backlog in order to maintain incentive compatibility. Specifically, the total reward for mining blocks must be non-decreasing. That is, after a block is mined, other miners must have enough transaction fees available in their mempools to be incentivized to move the chain forward instead of creating a fork that includes more fees. Ultimately, this motivates having a maximum block size limit that is small enough that miners cannot build blocks that claim "too much" of the outstanding transaction fees. This issue is discussed in more detail in Section 19.1.1.

10.2. Violating the Nakamoto Consensus Security Assumptions

Roughly speaking, there are three major ways that Nakamoto Consensus can fail. The following sections look at each of these concerns in turn:

1. The network delay is not sufficiently bounded such that blocks take "too long" to propagate.
2. The majority of the computing power deployed to the network can be malicious.
3. An adversary is able to create hash function collisions.

10.2.1. Network Delay and Block Propagation

Section 10.1 described how an adversary who can arbitrarily delay the transmission of blocks to honest miners would be capable of causing consensus failure. Whether adversarially induced or simply a result of typical network latency, any delay in block propagation is detrimental to Nakamoto Consensus. Specifically, latency in block propagation can cause the blockchain to fork. Two different miners may produce blocks at the same height of the chain at approximately the same time because they were both unaware of the competing block. Whenever forks occur, honest miners temporarily work on different problems, so the actual work that secures the chain is reduced.

The creation of new blocks is a Poisson process, which means that the times between two blocks being mined follow an exponential distribution. Thus, the probability of an unintentional fork in the blockchain is $1 - e^{\frac{-x}{T}}$, where x is the propagation delay and T is the expected block interval. For example, with Bitcoin's 600 second block interval and a 10 second propagation delay, there is approximately a 1.7% chance of conflicting forks appearing on the network when a block is found, so forks would be expected every 60 blocks on average. If the block interval were one minute instead, there would be more than a 15% chance of forks or forks every 6 or 7 blocks. According to an early study on Bitcoin's block propagation, it took 12.6 seconds for a block to propagate to 90% of the network [234]. However, that was prior to the deployment of techniques that reduced latency, such as compact blocks and the Fast Internet Block Relay Engine (FIBRE). Since then, accidental forks have become extremely rare.

The concerns regarding frequent forks relate to more than just maintaining consistency among nodes. Due to the permissionless nature of Nakamoto Consensus, adequate incentives are required for the system to work properly. When forks occur, only one of the competing blocks will actually remain in the chain for the long term, while the others are discarded and become stale, leaving their creators with no reward. Unfortunately, this implies that frequent forks can lead to mining power becoming concentrated among a smaller and smaller set of larger and larger entities. For example, consider a fork that occurs between a miner with 35% of the network's hash rate and a miner with only 5%. Both send their block out to the rest of the network and immediately begin mining on top of their own block. In this case, the larger miner instantly has an additional 30% of the network hash rate supporting their block compared to the small miner and thus has a 30% lower stale block rate. If it takes 30 seconds for a block to propagate to the rest of the network, there would be an approximately 4.88% chance of a conflict, so the larger miner gains an overall 1.46% revenue advantage. Difficulty adjustments make mining a low-margin business, so this revenue advantage can be substantial. In other words, big miners will create larger and larger blocks in order to squeeze out the competition over time (in the absence of a sufficiently small maximum block size limit).

Similar centralization issues arise based not just on the relative size of the miners but also on their relative network connectivity [235–237]. Miners who have low latency connections to other miners have the same types of advantages during forks that large miners do – the remaining miners are more likely to build off of the well-connected miner's block because they will see it first. In addition, better latency improves a miner's ability to include high fee transactions in their blocks before the transaction has fully propagated through the network.

A related concern is that miners are not strictly incentivized to get their blocks propagated to 100% of the network hash rate as quickly as possible but rather somewhere above 50% and less than 100% (see Section 9.4 on selfish mining, which is a related problem). This implies that crafting big blocks that reach the majority quickly but slowly propagate to the rest can harm small miners. It also implies that networking impediments like the Great Firewall of China are problematic. If the majority of the hash rate is on one side of the firewall, that side has a significant advantage over the miners on the other side. Miners can intentionally slow down block propagation in a few ways, such as stuffing blocks full of transactions that have not been forwarded to the rest of the network or including transactions in blocks that are deliberately slow to validate (see Section 19.2 for more information on slow to validate transactions).

Improving block propagation significantly reduces security issues with Nakamoto Consensus. One way of doing this is to increase the number of network connections that each node has, though that comes with the cost of increased bandwidth consumption [238]. There are also a variety of techniques that can be used to reduce block propagation time. Only compact blocks and the FIBRE relay network are described here because they are used on the Bitcoin network, though a variety of other block transmission mechanisms have been proposed (e.g., Xthin, Xthinner, Graphene, BlockTorrent, Falcon relay network, etc.). While important, these techniques have some limitations:

- They rely on the assumption that nodes have relatively synchronized mempools and perform poorly when this is not the case. That is, if there are many transactions that are included in a block but have not propagated to all of the miners, the extra network round trips and bandwidth of transmitting those transactions will reduce the effectiveness of compact blocks and increase latency and orphan rates. As block sizes increase and expected block intervals decrease, more transactions will exist that have not propagated fully.

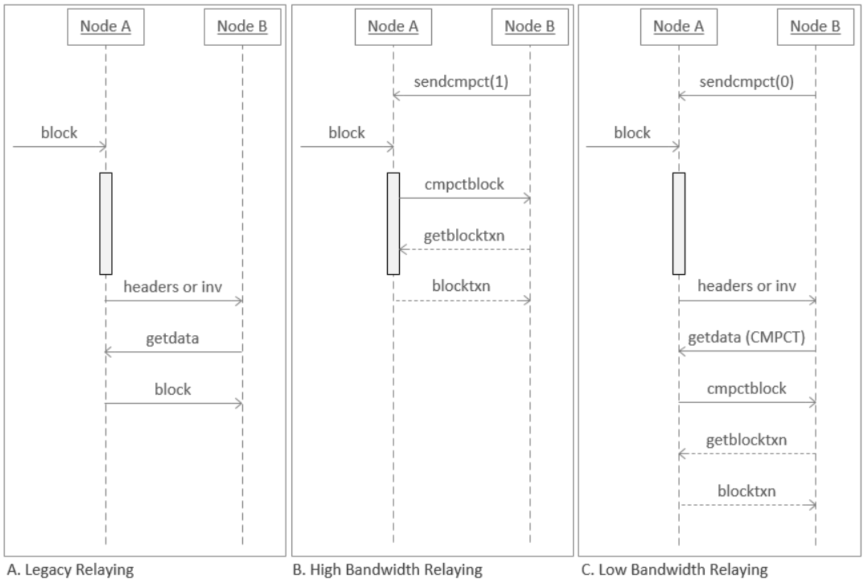


Fig. 23. Compact block message flow in which node A receives a block and sends it to node B. [239]

- While they may reduce latency and bandwidth for nodes that are currently online, nodes that were offline for a period of time or are being booted up for the first time must still download the complete block. Therefore, these techniques do not improve the performance of an initial synchronization and thus do not represent a complete scaling "solution."
- An adversary who does not want their block quickly propagated to all of the miners may use these techniques solely for personal advantage by, for example, using compact blocks to send blocks to a favored portion of the network and sending full blocks or nothing at all to a disfavored portion.

The compact blocks technique involves transmitting only the 80-byte block header, shortened transaction identifiers (txid) that are hardened to prevent denial-of-service attacks, the coinbase transaction, and a small selection of full transactions that the sending peer predicts the receiving peer may not have seen yet [239]. The receiving peer then tries to reconstruct the block themselves using the information provided and their own mempool and then requests any missing transactions. There's also a high-bandwidth mode where the receiving peer has a few of their peers send new blocks without asking first, which increases bandwidth (because the node may receive the block multiple times simultaneously) but reduces latency. The message flow can be seen in Figure 23. Typically, these can reduce what would have been a 1 MB block transmission to approximately 20 KB of data over the wire.

The short txids are created by taking a SHA-256 hash of the block header and nonce, using the fast hashing algorithm SipHash on the txid and some of the output of the SHA-256 hash, and removing the two most significant bytes of the result. The short txid is the remaining six bytes. Without this hashing procedure, it would be possible to create short txid collisions too easily, allowing attackers to prevent the scheme from working. While a 48-bit hash is not sufficient to prevent intentional collisions, the use of the block hash as a key to the hashing algorithm prevents an attacker from predicting the actual keys that will be used when the adversarially created transactions are included in a block. Additionally, peers have a 64-bit nonce that they share with each other and are unique per connection. These are mixed in to prevent even the block creator from controlling where collisions occur. The use of SHA-256 for this mixing is much slower than SipHash but is only performed once per block, so it does not add much overhead.

Another way that block propagation has been sped up is with the use of FIBRE,[1] which uses UDP instead of TCP and encodes (compact) blocks using forward error correction (FEC). TCP requires receiving the network packets that include the block in order, whereas UDP allows data to be consumed as fast as the network allows. FEC helps by better handling dropped packets. A FIBRE network is essentially an allowlist of miners who prioritize extremely low latency in block transmission. Because it consists of a curated or permissioned set of nodes, the network operator can theoretically perform censorship. However, anyone can set up a FIBRE network, so it is not conceptually dependent on a single centralized entity. Miners who utilize a relay network are substantially more likely to win in a block race than those who do not participate [240].

10.2.2. Majority Hash Rate Attacks (51% Attacks)

For Nakamoto Consensus (and other pure proof-of-work consensus mechanisms), a security assumption has been violated if the majority of the hash rate is malicious. An entity that controls the majority of the computational power deployed to the network has complete control over the content of blocks and can thus double-spend or censor transactions with certainty given enough time. This does not give the adversary the ability to change the rules arbitrarily, but it does allow them to decide which subset of valid transactions are added to the ledger and in which order.

One of the more unfortunate aspects of majority attacks is that they are largely self-funding. The majority attacker can replace any blocks created by honest miners with their own and thus claim 100% of the block rewards. Once an attacker acquires the majority of the computational power deployed on the network, incentive compatibility requires that the block rewards must be large relative to the benefits of attacking the chain. This condition is difficult to maintain unless "(i) the mining technology used to run the blockchain is both scarce and non-repurposable, and (ii) any majority attack is a ’sabotage’ in that it causes a collapse in the economic value of the blockchain" [241]. Stated differently, to dissuade a majority attacker, the proofs of work and the hardware used to create them must be useless outside of the network in question, and an attack must cause a decrease in the purchasing

---

[1] https://bitcoinfibre.org/

power of the underlying asset.

Part of the security argument against majority attacks is that a rational attacker is not going to harm the system that pays them. Of course, this does not hold in what is known as a *Goldfinger attack* – when an attacker expects to gain utility by causing the asset price to crash, perhaps because they have shorted the asset or are a central bank that fears currency competition.

This rationality argument is also inapplicable if a temporary majority can be created by renting hash rate. This is easier when the difficulty is low, perhaps because the network is still young or when the proof-of-work algorithm is ASIC-resistant and does not require specialized hardware. One strategy for renting computational power would be to form a negative fee mining pool that would have higher payouts than honest pools, which entices otherwise honest miners to the attacker's pool [242]. This is also possible using bribery via out-of-band payments, which require more trust, or *whale transactions* [243], which are high-value transactions intended to bribe miners onto a particular side of a fork. To create whale transactions, the attacker needs to first move some money to an address in the first block of their preferred fork so that spending from that address cannot happen on the other side of the fork. The attacker then creates a transaction with j outputs that are spendable by anyone but "timelocked" so that they can only be claimed in a series of j escalating blocks from the fork point. Miners who mine on this fork can then claim these bribes in the blocks they mine, but the bribes are never paid if the attacker's fork does not take over. As an alternative to this anyone-can-spend output method, a stream of high-fee transactions could work but would be less effective than the timelocks. Larger miners are more likely to switch to the whale transaction branch than smaller miners because the fork is then more likely to succeed, and larger miners may collect a higher proportion of blocks on the forked chain. It is also possible to use smart contracts deployed on other systems in order to facilitate bribery attacks [244–247].

In an extension to the model in [241], the ability for 51% attack victims to retaliate with their own rented hash rate or whale transaction bribes – which appears to have occurred on the Bitcoin Gold blockchain in February 2020 – disincentivizes majority attacks in the first place [248]. This result makes the relatively weak assumptions that the cost of the attacks and counterattacks increase over time and that the victim would suffer a cost in reputation for being the victim of a double-spending attack. This assumption is likely true for entities like cryptocurrency exchanges, which are also most likely to be the victims of majority attacks. However, this argument does not hold if the attacker is merely trying to censor transactions instead of double-spend. The simplest defense against censorship attacks is to have high transaction fees, which increases the opportunity cost of censorship to the attacker.

10.2.3. Hash Function Collisions

If there were efficient ways to produce hash collisions, then proof of work would not be effective as a Sybil-resistance mechanism, and consensus would trivially fail. In addition, the hash references to previous blocks in the blockchain would no longer commit to the entirety of the chain up to that point and would fail to properly commit to the data in the blockchain itself. For example, an adversary capable of finding hash collisions would be able to arbitrarily change the content of blocks at any point in the blockchain without actually breaking the chain itself and could thus arbitrarily manipulate the state of the system to their advantage. As a consequence, none of the data in the blockchain would be tamper-resistant or have any integrity, and it would be impossible for nodes to agree on the state of the system.

10.3. (More) Attacks Against Nakamoto Consensus

Even when the typical security assumptions for Nakamoto Consensus hold, there are still possible attacks against users of the system. In particular, attackers with less than half of the network's hash rate can still attempt to double-spend or censor transactions. The Bitcoin white paper [1] considered double-spending with less than half the computational power assuming a fixed hash rate and constant difficulty, although the calculations were flawed and later corrected by Rosenfeld in [249]. The updated probabilities can be found in Figure 24.

The first step of the attack is to mine a block that includes a transaction in which the attacker sends some of their funds to themselves. They withhold this block from the network and then broadcast a transaction that sends those same funds to a merchant. The merchant waits for k confirmations before giving the attacker whatever they purchased, and the attacker continues mining on their secret branch of the chain during this time. If the attacker manages to build a longer chain, they broadcast it to the network, which then adopts that chain as the best one. This overwrites the transaction sending funds to the merchant, and the attacker can walk away with both their original funds and the goods they did not pay for.

A piece of Bitcoin folk wisdom is that merchants should wait for six confirmations before accepting a transaction. This number was selected by bounding the risk of a double-spend to 0.1%, assuming that the attacker has amassed no more than 10% of the network's hash power. However, these numbers are arbitrary, and as the attacker's computational power gets closer to 50%, the number of required confirmations blows up toward infinity. On the other hand, small valued transactions are likely safe after just one or two confirmations. To be profitable (rather than just probable), the attacker should incorporate a stopping threshold for when their private chain falls too far behind and success is unlikely [250].

The attack can also be run multiple times concurrently to steal from multiple merchants at once. This makes it harder for a merchant to calculate how long to wait based on the value

q	1	2	3	4	5	6	7	8	9	10
2%	4%	0.237%	0.016%	0.001%	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
4%	8%	0.934%	0.120%	0.016%	0.002%	≈ 0	≈ 0	≈ 0	≈ 0	≈ 0
6%	12%	2.074%	0.394%	0.078%	0.016%	0.003%	0.001%	≈ 0	≈ 0	≈ 0
8%	16%	3.635%	0.905%	0.235%	0.063%	0.017%	0.005%	0.001%	≈ 0	≈ 0
10%	20%	5.600%	1.712%	0.546%	0.178%	0.059%	0.020%	0.007%	0.002%	0.001%
12%	24%	7.949%	2.864%	1.074%	0.412%	0.161%	0.063%	0.025%	0.010%	0.004%
14%	28%	10.662%	4.400%	1.887%	0.828%	0.369%	0.166%	0.075%	0.034%	0.016%
16%	32%	13.722%	6.352%	3.050%	1.497%	0.745%	0.375%	0.190%	0.097%	0.050%
18%	36%	17.107%	8.741%	4.626%	2.499%	1.369%	0.758%	0.423%	0.237%	0.134%
20%	40%	20.800%	11.584%	6.669%	3.916%	2.331%	1.401%	0.848%	0.516%	0.316%
22%	44%	24.781%	14.887%	9.227%	5.828%	3.729%	2.407%	1.565%	1.023%	0.672%
24%	48%	29.030%	18.650%	12.339%	8.310%	5.664%	3.895%	2.696%	1.876%	1.311%
26%	52%	33.530%	22.868%	16.031%	11.427%	8.238%	5.988%	4.380%	3.220%	2.377%
28%	56%	38.259%	27.530%	20.319%	15.232%	11.539%	8.810%	6.766%	5.221%	4.044%
30%	60%	43.200%	32.616%	25.207%	19.762%	15.645%	12.475%	10.003%	8.055%	6.511%
32%	64%	48.333%	38.105%	30.687%	25.037%	20.611%	17.080%	14.226%	11.897%	9.983%
34%	68%	53.638%	43.970%	36.738%	31.058%	26.470%	22.695%	19.548%	16.900%	14.655%
36%	72%	59.098%	50.179%	43.330%	37.807%	33.226%	29.356%	26.044%	23.182%	20.692%
38%	76%	64.691%	56.698%	50.421%	45.245%	40.854%	37.062%	33.743%	30.811%	28.201%
40%	80%	70.400%	63.488%	57.958%	53.314%	49.300%	45.769%	42.621%	39.787%	37.218%
42%	84%	76.205%	70.508%	65.882%	61.938%	58.480%	55.390%	52.595%	50.042%	47.692%
44%	88%	82.086%	77.715%	74.125%	71.028%	68.282%	65.801%	63.530%	61.431%	59.478%
46%	92%	88.026%	85.064%	82.612%	80.480%	78.573%	76.836%	75.234%	73.742%	72.342%
48%	96%	94.003%	92.508%	91.264%	90.177%	89.201%	88.307%	87.478%	86.703%	85.972%
50%	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Fig. 24. Probability of successful double spend as a function of the attacker's hash rate q and number of confirmations. These probabilities assume that the attacker has already mined and withheld a single block that includes the self-paying double-spend transaction. [249]

of a transaction, and they may conservatively have to consider the value of an entire block. Further, an attacker with arbitrarily low hash rate can profitably implement a double-spend by combining it with selfish mining. In this case, the attacker mines in secret for a double-spend, but when it is unlikely that they will be able to reorg a sufficient number of blocks, they switch to selfish mining and publish their secret blocks to get the block rewards.

Sometimes, for small transactions, a merchant may accept a transaction without any confirmations (colloquially called "0-conf"), even though the protocol provides no security guarantees for these transactions even against adversaries that do not control any computational power. Assuming that most spenders are honest, this might be acceptable from a risk management perspective. However, when the attacker does control computational power, additional attacks are possible, including the *Finney attack* and *vector76 attack*. Both of these attacks can be generalized in order to attack merchants who require confirmations as well [251].

The Finney attack was proposed by cryptographer Hal Finney and starts with the attacker mining a block that contains a transaction sending their funds to another address that the attacker controls. Instead of broadcasting the block to the network, the attacker broadcasts a transaction from the original address to the merchant, collects the goods from the merchant, and finally broadcasts the pre-mined block to double-spend [252]. The Finney attack can also be used when the merchant requires *k* confirmations before accepting a transaction. In

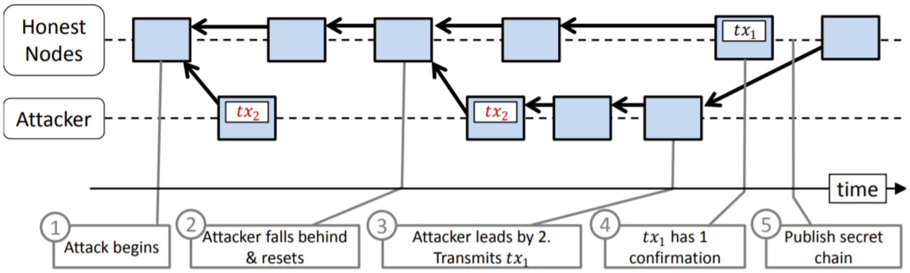


Fig. 25. Finney attack example. [251]

this case, the adversary works on a secret chain that embeds the double-spend transaction, tx_2, which conflicts with the merchant transaction, tx_1. The attacker attempts to create $k+1$ blocks more than the honest miners but aborts if the honest miners ever get ahead of the attacker, potentially restarting at a later block in the chain. Once the attacker has a $k+1$ block lead, they broadcast tx_1, wait for the honest miners to produce enough blocks for tx_1 to have k confirmations, and then broadcast their private chain to overtake the network and reverse tx_1. An example of a 1-conf generalized Finney attack appears in Figure 25. Importantly, if the attacker can choose when they want to submit a transaction, the pre-mining stage can be attempted repeatedly until the attack can be executed with a near-guarantee of success.

The vector76 attack requires the adversary to have a direct connection to the victim [253]. The attacker mines a block that includes a transaction sending funds to the merchant but does not broadcast the block or the transaction to the rest of the network. The attacker then waits until another block is mined at the same height by the honest miners before sending the pre-mined block directly to the victim (hopefully before the victim sees the other block) and collecting the goods. Finally, the attacker broadcasts a conflicting transaction that refunds themselves. The network adds this double-spend transaction to a block, and the original pre-mined attack block becomes stale.

There is a generalized version of this attack that works against victims that do not relay blocks they have received, like light clients [251]. This is one reason why light clients are less secure than full nodes and why merchants using light clients should wait for more confirmations than they otherwise would. For example, say that a merchant only uses a light client for validation and requires k confirmations to accept a transaction. The attacker starts pre-mining a secret branch that includes a transaction to the merchant, tx_1, in the first block of their private chain. They continue mining on this branch until tx_1 has k confirmations. If the attacker's branch is longer than the honest branch, they show the merchant their pre-mined chain, collect the goods, and stop mining on the private branch. Finally, the attacker broadcasts a conflicting transaction, tx_2, to the network, which is unaware of the attack chain that is known only to the attacker and the merchant. Eventually, the honest network will build a longer branch than the private one that includes tx_2, thus reversing tx_1. Figure 26 shows an example of this attack executed against a merchant requiring two

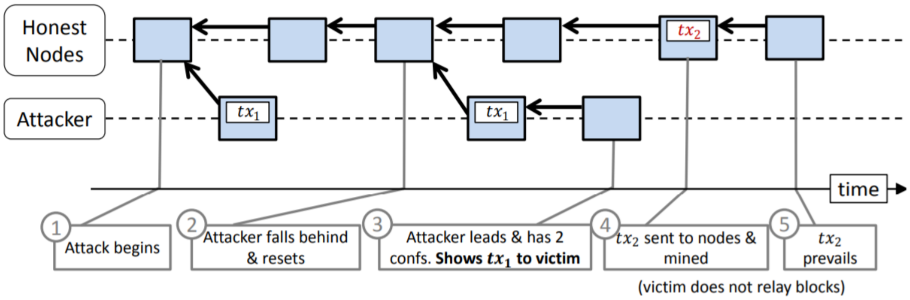


Fig. 26. Vector76 attack example. [251]

confirmations.

So far, this section has only considered ways in which a minority hash rate attacker can double-spend against adversaries who require only a small number of confirmations. An adversary could also attempt to censor transactions via *feather-forking* [254]. In a feather-forking attack, the miner with a minority of the hash rate commits to not mining on top of any chain that contains an address or transaction that they would like to censor, at least for several blocks. If the attacker makes it credibly known to other miners that they are following this policy, then other miners are less likely to include the address or transaction in question for fear that their block might be reorged. If the attacker is able to coax more than half of the total hash rate to censor, they have succeeded.

11. More Proof-of-Work Protocols

Nakamoto Consensus is the paradigmatic proof-of-work consensus algorithm, but many others have been proposed or implemented. This section begins by considering some more modest adjustments to Nakamoto Consensus. Most of these protocols attempt to address one of its primary flaws: the degradation of security that occurs when block transmission latency is too high. Several also attempt to improve upon its low chain quality in order to resist selfish mining attacks. The GHOST fork-choice rule is used to allow blocks to be produced more quickly and reduce the time until transaction settlement. FruitChains modify the blockchain structure in order to achieve better chain quality guarantees.

Other protocols run Nakamoto Consensus over multiple separate blockchains in parallel, which can substantially reduce the latency of transaction settlement. By giving these blockchains different roles, the Prism construction is able to increase throughput and improve latency. Several protocols utilize DAGs instead of linear blockchains. These approaches address Nakamoto Consensus's block transmission issue by largely doing away with the concept of stale blocks in the first place and allowing all competing blocks to ultimately be incorporated into the ledger. This comes with increased complexity in protocol design and analysis and, in some cases, only results in a partial rather than total ordering of transactions, making them unsuitable for generic smart contracts.

Finally, there are protocols that use proof of work as a Sybil-resistance mechanism to control who may participate in classical permissioned BFT consensus. These protocols allow responsive transaction commitment, where settlement occurs at the actual speed of network communication rather than based on the maximum network delay. This advantage comes at the cost of increased trust assumptions that are more in line with proof-of-stake protocols than the other proof-of-work ones discussed here.

Note that many of the protocols presented in this section have a proof-of-stake "equivalent" or can be used as subprotocols in proof-of-stake systems. The security ramifications would need to be considered in their respective environments, but the protocols are included here because they were originally proposed and analyzed in the proof-of-work context. This is the case for GHOST, FruitChains, Parallel Chains, and several of the DAG protocols.

11.1. Nakamoto Consensus Protocol Adjustments

Recall that the primary security challenge with Nakamoto Consensus relates to the network latency of block transmission (see Section 10.2.1). A variety of algorithms attempt to work around this limitation.

11.1.1. Weak Blocks and Pre-Consensus

Weak blocks are a way to slightly modify Nakamoto Consensus in order to help synchronize mempools between miners. The idea is that there are two separate proof-of-work difficulty targets: the typical one for mining blocks and a substantially easier one for mining weak blocks (analogous to mining pool shares). For example, the weak block difficulty may be set to $\frac{1}{10}$ of the normal block difficulty such that, in expectation, there are 10 weak blocks found in between normal blocks.

Like normal blocks, weak blocks carry transactions. This helps miners know which transactions other miners have seen in the meantime. Essentially, blocks are transmitted piece by piece during the standard block interval rather than all at once in a burst when blocks are found. This helps minimize block propagation delays and allows for higher throughput without increasing the risk of stale blocks. Weak blocks also allow "pre-consensus" on transactions that have yet to be confirmed, slightly increasing the security of accepting transactions with no confirmations. Weak blocks require additional overhead and are only effective if the majority of miners participate. Further, non-mining full nodes may not be incentivized to broadcast weak blocks because they use bandwidth but only benefit miners directly.

Specific weak block proposals include Subchains [255] and Flux [256]. The Subchains idea is merely an addition to Nakamoto Consensus, whereas Flux actually modifies the consensus algorithm. The Subchains protocol works as follows, starting from the point where a miner sees and accepts a regular block:

1. The miner begins working on the next block, including a hash pointer to the block that

was just accepted, and finds a proof-of-work that satisfies the weak block difficulty. The miner broadcasts this weak block to the network.

2. Miners verify the weak block. If valid, miners adjust the coinbase transaction's reward to include the new transaction fees, add any new transactions to their block candidate that they desire, and compute a new Merkle root for the block header. They then continue looking for a valid nonce to find another proof of work. Denote the new coinbase transaction, the newly included transactions, the Merkle root, and the previous block hash as a Δ-*block*.
3. When miners compute new proofs of work that create weak blocks, they broadcast only the Δ-block and the hash of the previous weak block. That is, miners essentially cooperate to build the next block incrementally by building a subchain.
4. Eventually, a miner will find a regular block that satisfies the more difficult target. When they do so, they broadcast only their Δ-block and the hash of the previous weak block. Note that this should require significantly less bandwidth than broadcasting a full block, thus improving latency.

If there are multiple competing subchains, honest miners build on top of the longest subchain that they are aware of. If conflicting transactions exist, any transaction included in a subchain is prioritized over competing transactions that only exist in their mempool.

Flux modifies the fork-choice rule to include the work from weak blocks in the total work when deciding which chain is "longest" so that a lengthy chain of weak blocks can reorg a normal block. Flux also changes the incentives to include revenue sharing among those who mine weak blocks. When building a subchain of weak blocks, miners must include the prior weak block miner addresses in their coinbase transaction in order to be considered valid, proportional to the number of weak blocks found. This revenue-sharing scheme reduces the variance of the mining reward and may reduce the profitability of selfish mining. On the other hand, miners who have already created weak blocks in the subchain lose rewards when new weak blocks are added, so they may try to create forks in the subchain or not propagate other miners' weak blocks. Additionally, once a long subchain exists, miners may no longer find it profitable to mine until the next regular block is found.

11.1.2. Bitcoin-NG

Bitcoin-NG ("next generation") was one of the earliest proposals for modifying Nakamoto Consensus to reduce block transmission latency [257]. The primary idea is to separate the leader election process from transaction serialization, which is achieved by having *key blocks* and *microblocks*. Key blocks are like normal Bitcoin blocks from a consensus perspective but do not include transactions and that specify a public key for the leader. Microblocks are produced between key blocks and contain transactions. Mining a key block makes the public key holder the leader of the next epoch, and they are entitled to mine microblocks at a fixed rate (microblocks do not require proof of work but rather are signed by the specified public key). Bitcoin-NG uses the longest chain rule for the chain of key blocks but changes the incentives: the block subsidy goes to whomever finds the key block, but transaction fees are split so that 40% goes to the leader (and thus microblock producer) and 60% goes to the miner who finds the next key block.

Microblocks do not include a proof of work, so a malicious leader can split the chain (of microblocks) for free. To prevent this, Bitcoin-NG uses *poison transactions* to invalidate the rewards of malicious leaders. Poison transactions include the header of the first microblock of a conflicting fork to prove that fraud has occurred. This special transaction can be included in any key block during the coinbase maturity window of the malicious leader's key block and rewards the leader who includes it with a fraction of the forfeited funds.

Because key blocks do not include transactions, they are small and should propagate through the network quickly, so increasing transaction throughput does not increase the risk of a key block becoming stale. However, there are a number of trade-offs, including making SPV light clients impractical, eliminating the ability of individual miners to choose which transactions are in a block (only mining pool leaders can do so), and potentially encouraging denial-of-service attacks against leaders by revealing their identity before they have produced microblocks.

Several works have revisited Bitcoin-NG's incentives [258, 259]. In [258], it is shown that the reward split should provide only $\frac{3}{11}$ of the reward to the leader instead of 40% because the original analysis ignored the possibility that a miner in one epoch also gets the key block for the next epoch. Several additional selfish mining-related attacks are shown in [259], and the microblock architecture increases the profitability of selfish mining when the attacker has more than 35% of the network's hash rate.

11.1.3. Tie-Breaking Schemes

When multiple blocks are mined simultaneously, miners must decide which block to mine on top of. In Bitcoin, nodes accept the first block they see until one side of the fork has more work than the other. In Ethereum and Bitcoin-NG, nodes choose uniformly at random which side of the fork to prefer (UTB). In the following sections, additional schemes called DECOR+ and Publish or Perish (PoP) suggest other tie-breaking methods (DECOR+ is discussed in more detail in Section 11.1.4, and PoP is detailed in Section 11.1.5). An early DECOR+ proposal broke ties using the chain tip with the smallest hash (SHTB) [260]. The updated DECOR+ uses an unpredictable deterministic tie-breaking scheme (UDTB) where the winner is chosen based on a pseudorandom function taking all competing blocks as inputs [261]. PoP compares chain "weights," and blocks published after their competitors do not contribute weight, while blocks that incorporate links to their parents' competitors have higher weights. Thus, blocks that are kept secret until competing blocks are published will contribute to neither or both branches and thus confer no advantage in winning the block race [262].

Each of these choices has its own security ramifications, particularly with respect to selfish mining resistance (see Section 9.4 for background). By accepting the first seen block, Bitcoin allows the honest computational power to become split such that sufficiently well-connected adversaries can selfish mine profitably at arbitrarily low hash rates. However, if the attacker is not well-placed on the network, it is the tie-breaking method most resistant to selfish mining. UTB and UDTB have nearly identical resistance to selfish mining, and neither outperforms Nakamoto Consensus when $\gamma \leq 0.5$. By not taking into account the time that a block was received, both UTB and UDTB allow an attacker who mines "from behind" to still win the block race with a tie [263]. SHTB has the lowest chain quality of all because a miner who finds a block with a particularly low hash can continue selfish mining privately and feel confident that they will win block races when they occur, so they can strategically deviate when the odds are in the attacker's favor. SHTB also suffers from the same mining from behind issue as UTB and UDTB. When $\gamma = 0$, none of these protocols have better chain quality than Nakamoto Consensus for $\alpha < 0.39$. However, PoP begins to outperform Nakamoto Consensus when $\alpha \geq 0.4$ [263]. The poor chain quality of Nakamoto Consensus and these other tie-breaking variants is caused by an information asymmetry, where the attacker has more information than the honest miners regarding network connectivity [263].

11.1.4. DECOR+

DECOR+ (deterministic conflict resolution) is a revenue-sharing mechanism intended to incentivize miners to converge on the same block during block races [260, 261]. It tries to share the reward among all miners who mined a block at the same height (uncles) in order to allow faster block production. DECOR+ was designed such that if all nodes have access to the same blockchain state, any conflicts will be quickly resolved in a deterministic fashion that is incentive-compatible for miners [260]. This prevents honest miners from splitting their hash rate between two forks for long. Originally, it was supposed to break ties using the block with the highest fees and then the smallest hash if fees were tied, but fees are gameable by miners (e.g., they can include their own transactions with arbitrary fees paid to themselves, and users can also pay fees to miners out-of-band). In [261], a different selection function was proposed: hash all block headers and then take the XOR-sum, modulo the number of competing blocks. This way, the miner cannot compute their block in a way that gives them a higher chance of winning.

A number of possible reward functions were described in [261], each of which accounts for uncle blocks in order to incentivize the inclusion of uncle references in blocks. To bound the maximum money supply, the reward function can bound the number of uncles for which a reward is provided. For instance, the reward function may put the L competing block headers in order by block hash and reward the N lowest ones. There can also be a punishment fee if a miner does not follow the deterministic selection function. For example, let X be the deterministically selected block, Z the block selected by the miner, Y the mined block, r the total block reward, i the inclusion reward per uncle, and p the punishment fee.

If $X \neq Z$, let $r_y = \frac{r(1-p)}{N+1}$. Otherwise, let $r_y = \frac{r}{N+1}$. Then the reward array is:.

$$[r_{i_1}, \cdots, r_{i_N}, r_{i_{N+1}}, \cdots, r_{i_{N+L}}, R_Y] := [\frac{r}{N+1}, \cdots, \frac{r}{N+1}, 0, ..., 0, r_Y + iN] \tag{4}$$

By punishing miners who deviate from the deterministic block of choice, DECOR+ is more resistant to selfish mining and double-spending but at the cost of being more susceptible to feather-forking censorship [263]. This is because a malicious miner has an easier time decreasing the income of honest miners.

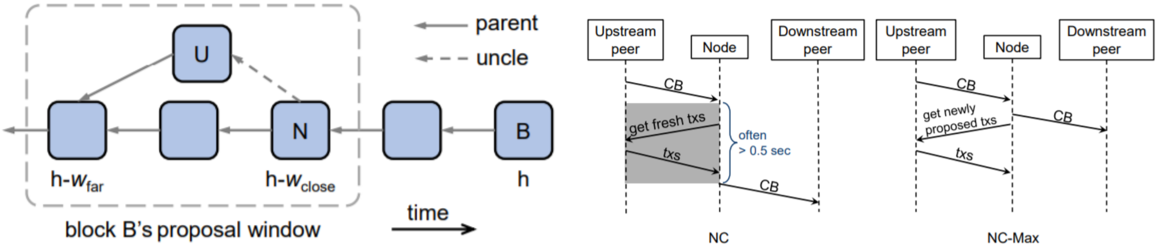
11.1.5. Publish or Perish

Publish or Perish (PoP) was proposed as a defense against selfish mining and makes it so that withholding blocks does not provide a miner with an advantage in winning block races [262]. In this scheme, a node will consider a block *in time* if the height of the block is larger than the node's local best chain height or if its height matches the local best chain height and is received within a bounded length of time (corresponding to a typical network delay) from seeing the first block of that height. PoP uses a nonstandard definition of "uncle" blocks, where an "uncle" must be in time, and the height of a block's "uncle" must be one less than the height of the block. Miners are encouraged to include references to these "uncles" in the blocks that they mine.

The fork-choice rule nodes follow in PoP uses the *weight* of a chain, which is the number of in-time blocks plus the number of in-time "uncles" referenced in the in-time blocks. It also uses a security parameter k (where $k = 3$ is suggested by the authors), which manages a trade-off between selfish mining resistance and the ability of nodes to quickly recover from network partitions. Specifically, when a block race occurs, the fork-choice rule is as follows:

1. If one chain is longer than another by at least k blocks, then follow the longest chain.
2. If the difference is less than k blocks, then follow the chain with the most weight.
3. If chains are tied for highest weight, then follow a random one.

If a selfish miner withholds a block and keeps it secret until after a competing block has already been published, then the selfish miner's block will not contribute to the weight of the attacker's chain. Alternatively, if the attacker's block is published around the same time as the competing honest block, then the next honest block that is produced can include an "uncle" reference to the previously secret block, which increases the weight of the honest chain. In either case, the selfish miner does not gain an advantage in the block race by withholding. Because of the trade-off between selfish mining resistance and tolerating network partitions, merchants in a system that uses the PoP fork-choice rule may want to wait at least k blocks before considering a transaction of non-trivial value sufficiently confirmed.



(a) NC-Max proposal window **(b)** Block propagation in NC vs. NC-Max

Fig. 27. NC-Max block propagation mechanism. Panel (a) shows the NC-Max proposal window with $w_{far} = 4$ and $w_{close} = 2$. If a transaction is proposed in block U and committed in B, the transaction fee goes to the miner of block N. Panel (b) shows how compact blocks can propagate more quickly using NC-Max compared to Nakamoto Consensus. [264, 265]

11.1.6. NC-Max

NC-Max attempts to remove the latency-related scalability bottleneck of Nakamoto Consensus by separating transaction synchronization from transaction confirmation [264, 265]. It builds off of the compact blocks idea (see Section 10.2.1) and attempts to work around the issue of requiring more round-trip communication to handle transactions that have not propagated through the network when blocks are found. It does so via a two-step process of transaction proposal and transaction commitment.

In NC-Max, miners include references to uncle blocks (stale blocks in the same difficulty epoch) within the blocks that they find. The part of the block that contains transactions is called the transaction commitment zone. Each block also has a *transaction proposal zone* that includes txpids, or the first few bytes of the transaction ID, and a transaction is considered proposed if it is included here (or if it was proposed in an uncle block that was referenced). The transactions referenced in the transaction proposal zone need not be valid transactions for the block itself to be valid. A new validity rule is added for transactions in the commitment zone at height h: it must have been proposed during the *proposal window* in a block of height $h - w_{far}$ to $h - w_{close}$. An example is shown in Figure 27a. Note that the coinbase transaction is excluded from this mechanism.

Nodes forward compact blocks that include the transactions in the proposal zone to their peers once they themselves have reconstructed the commitment zone. They should have all of these transactions after receiving the blocks in the proposal window. In the meantime, the node requests any proposed transactions they have not yet seen from their peers. As a result, the extra round-trips required to receive these transactions do not impact block propagation time. A comparison of compact block propagation in Nakamoto Consensus and NC-Max is in Figure 27b.

NC-Max also adjusts the reward distribution and the difficulty adjustment algorithm. When a transaction is committed, the fee is split 70%-30% between the miner who commits it and

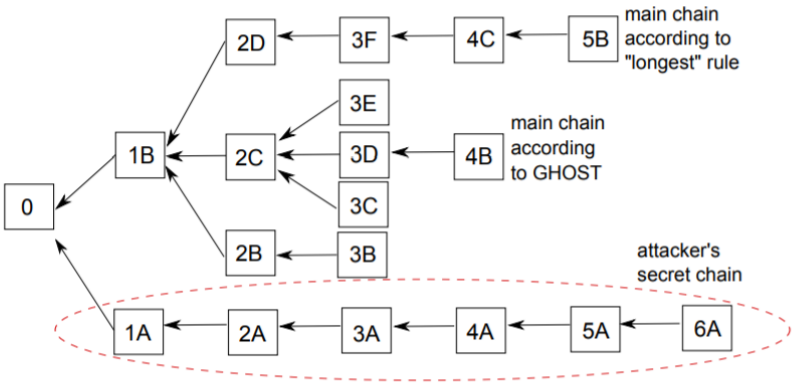


Fig. 28. GHOST fork choice rule, assuming all blocks have the same difficulty. An attacker chain would overtake the "longest" chain but not the GHOST chain, which consists of blocks 0, 1B, 2C, 3D, and 4B. In this example, the subtree that begins at block 1B has 12 blocks, whereas its competitor at 1A only has six. The subtree of block 2D has four blocks, for 2C has five blocks, and for 2B has only two blocks. The subtree beginning at block 3D has two blocks, as opposed to only one block for 3E and 3C. Every block added to the subtree of block 1B, regardless of block height, contributes to the security of block 1B. [266]

the miner who proposes it [265]. Uncle miners do not collect any reward. The details of the difficulty adjustment are out of scope for this document but take into account the uncle blocks from the prior epoch in order to target a pre-specified stale block rate.

11.2. Greedy Heaviest-Observed Sub-Tree (GHOST)

One of the earliest proposed fork-choice rules aside from the longest chain rule was GHOST, a variant of which was later used in the Ethereum network [266]. It allows the expected block interval to be much shorter than Nakamoto Consensus because the proofs of work used for stale blocks are counted when deciding which chain is canonical. In other words, blocks that do not make it into the main blockchain still contribute to the total work for the chain. Contrast this with Nakamoto Consensus, where stale blocks contribute nothing to the security of the chain.

In a GHOST blockchain, blocks contain an extra field that is used to reference uncle blocks. This creates a DAG of blocks and block references but still only chooses a linear chain of blocks by using the information contained in the DAG. When a fork exists, the GHOST algorithm greedily selects the heaviest subtree of blocks that begin from any fork point, starting from the genesis block. See Figure 28 for an example of the GHOST fork-choice rule. As one may infer from the example, GHOST leads to a small weakening in the chain growth property compared to Nakamoto Consensus, but this does not adversely impact the security of the chain.

GHOST has been proven secure in a synchronous network with constant difficulty and has superior liveness (when not under attack) though worse chain quality compared to

Nakamoto Consensus [267]. The consistency property of GHOST was also proven in [230], but contrary to many peoples' expectations, it has the same consistency bounds as Nakamoto Consensus. Therefore, GHOST is unable to handle a significantly higher throughput with the same security from a consistency perspective. It will degrade in security just as Nakamoto Consensus does (though it can still tolerate shorter block intervals).

GHOST's liveness can suffer due to an attack that requires less than half of the network's computational power [230]. The adversary attempts to maintain forks for as long as possible and thus delay transaction confirmation. When a fork occurs, the adversary attempts to keep both subtrees balanced by using their computational power on whichever side of the fork requires it. When the block interval is low compared to block propagation time, [230] showed that forks can be maintained for 10 or more blocks with non-negligible probability. GHOST is more susceptible to this attack because blocks mined by the adversary can be withheld from the network for longer than they can in Nakamoto Consensus and still contribute to the subtree as uncle blocks.

Due to its use in Ethereum, GHOST has been subject to more real-world testing than any fork-choice rule other than the longest chain rule of Nakamoto Consensus. However, because it is not as simple of a rule, it makes other aspects of the system – such as incentivization and the choice of difficulty adjustment algorithm – more complicated and less well-studied. This can lead to security issues. For instance, Ethereum's variant of GHOST provides incentives for miners of uncle blocks, and an old version of the protocol allowed a strategy that would create undesirable inflation and allowed greedy miners to gain at the expense of coin-holders [268]. This issue resulted in a change to Ethereum's difficulty adjustment algorithm. Additionally, GHOST with uncle rewards is more susceptible to selfish mining than Nakamoto Consensus [210–212]. As α increases, the revenue of both selfish and honest miners increases due to uncle rewards, which may lead to greater inflation of the supply of ether. Rewarding uncle blocks may also encourage block withholding and fork-after-withholding attacks [269, 270].

11.3. FruitChains

Unlike GHOST, which reduces chain quality, the FruitChains protocol was designed in order to improve chain quality and reduce the efficacy of selfish mining [271]. Specifically, FruitChains are δ*-approximately fair*: any honest miners that control a ϕ fraction of computational power are guaranteed (with high probability) to get at least a $(1-\delta)\phi$-fraction of the rewards in any $\Omega(\frac{\kappa}{\delta})$ length portion of the chain, with κ as a security parameter. This prevents any adversarial minority of the hash rate from improving their revenue more than a factor of $(1+3\delta)$ compared to honest mining.

The FruitChains protocol has miners simultaneously mine normal blocks as well as *fruits*, which have a lower difficulty. As with Nakamoto Consensus, miners follow the chain of blocks with the most work. However, unlike Nakamoto Consensus, blocks contain fruits instead of transactions, and the fruits are where transactions are recorded. A fruit must be

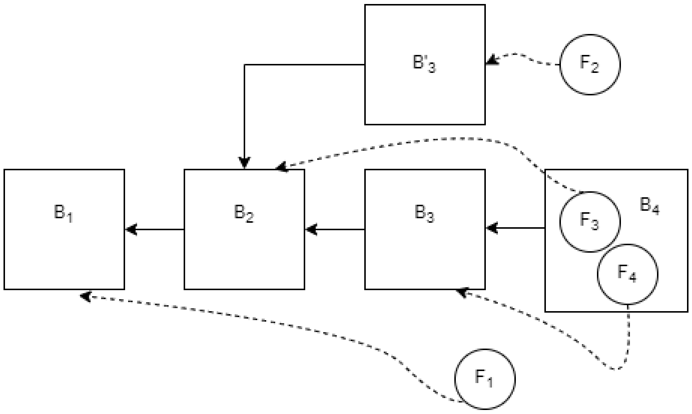


Fig. 29. FruitChains architecture. Assume the recency parameter $T_0 = 3$. In this case, fruits F_3 and F_4 are valid fruits included in block B_4, because they point to only the two most recent blocks. Fruit F_1 could not be included in block B_4, because it isn't sufficiently recent. Fruit F_2 also cannot be included, because it points to a stale block.

sufficiently "recent" to count; that is, it must include a reference to a block not too far in the past from the block that contains the fruit (say, within T_0 blocks). This prevents fruits mined long in the past from being included. It also prevents fruit-withholding attacks, where the attacker builds up a bank of fruits and then mines them all into the chain in a short period of time, causing a large fraction of fruits to be adversarial. To construct a linearized log of transactions, an ordered sequence of distinct fruits must be extracted from the chain, including only the first fruit if duplicates exist. Then the transactions are extracted in order from this chain of fruits, also removing duplicates as needed. The block reward and transaction fees for a block are evenly distributed to the miners of a constant-length number of blocks preceding the block in question. The FruitChains architecture is shown in Figure 29.

The simultaneous mining of fruits and blocks utilizes a "2-for-1" proof-of-work technique described in [9], which allows a singe random oracle $H()$ to operate as two independent oracles, $H_0()$ and $H_1()$. In other words, miners can attempt to compute proofs of work for two different schemes for the cost of one oracle query without the ability to use their compute power to favor one process over the other. An example would be to use the m most significant bits of the hash function's output for one scheme (e.g., mining blocks) and the n least significant bits for another (mining fruits).

A detailed security analysis of FruitChains was provided in [263]. In FruitChains, mining a block provides no direct reward (except a share of the rewards from future blocks), which has significant security implications. For example, a selfish miner has no incentive to publish blocks when neither the secret chain nor the honest chain have reached T_0 blocks since the fork point – the attacker's fruits that were mined before the T_0-th block – will also appear in the honest chain. If the attacker is able to win a block race of at least T_0 blocks, they can invalidate all honest fruits, so double-spending is more profitable against FruitChains than against Nakamoto Consensus. Increasing T_0 makes this more challenging and reduces the incentive to selfish mine, but the safe transaction confirmation delay increases linearly with T_0. By making fruit mining increasingly less difficult compared to block mining (a larger fruit-to-block ratio), selfish mining becomes less profitable but at the expense of having more repeated transactions and fruits that need to be removed from the ledger, thus wasting bandwidth. On the positive side, FruitChains are more censorship-resistant than Nakamoto Consensus because the attacker would need to overwrite T_0 blocks to invalidate honest fruit.

11.4. Parallel Chain Approaches

One way to reduce the latency of transaction settlement is to run multiple blockchains in parallel and then use some procedure to combine the contents of the separate chains [272, 273]. To that end, [272] generalizes the "2-for-1" mining technique of [9] (and described in Section 11.3) into an m-for-1 mining scheme. Then, given an underlying blockchain protocol, the technique can be used to execute m instances of it in parallel while the same mining operation is used across all m instances without allowing a deviant miner to focus their computational power onto a particular chain. A generic technique for then deterministically combining these m (nearly) independent ledgers into a single *virtual ledger* was described in [273]. Care must be taken when handling the design of the difficulty adjustment algorithm for schemes that employ parallel chains because naive adaptations of common DAAs are insecure in this context [274].

Nakamoto Consensus has relatively high transaction settlement latency due to the need to wait for enough blocks to confirm a transaction and stabilize an agreed-upon chain prefix, where the possibility of a common prefix violation decreases exponentially in the number of blocks. This settlement latency is directly related to and limited by the latency of block propagation on the underlying network. By combining m ledgers like this in parallel, settlement time can be reduced by up to a $\Theta(m)$ multiplicative factor when including a transaction in each ledger. This is done by using a ranking algorithm for each underlying ledger and then combining these ranks by an exponential sum of the ranks of the transaction from each individual chain. This allows for a trade-off between transaction fees and settlement time: clients can issue a transaction on a single ledger and pay one transaction fee with settlement times roughly on par with the settlement time of the underlying blockchain or alternatively pay up to m transaction fees to achieve the multiplicative $\Theta(m)$ reduction in settlement time.

The details of how this ranking and combining are performed are beyond the scope of this document, but roughly, the ranking for a blockchain using Nakamoto Consensus could be the timestamp embedded in the block that contains the transaction in question. The combined rank is then essentially an average of the ranks in the individual blockchains, which amplifies the exponential rate at which transactions are settled:

$$e^{-\frac{combinedrank(tx)}{L}} = \frac{1}{m} * \sum_{i=1}^{m} e^{-\frac{rank_i(tx)}{L}}, \tag{5}$$

where L is a parameter that is proportional to the security parameter of the system.

The following subsection presents the Prism algorithm, which is a concrete instantiation of a parallel chains approach rather than the abstract and generic one presented here.

11.4.1. Prism

Unlike the generic parallel chains construction discussed above, Prism optimizes throughput in addition to latency [275, 276]. It is capable of achieving optimal throughput by taking advantage of the network's full communication bandwidth and near-optimal transaction settlement latency of approximately the network propagation delay.

In a longest chain protocol, blocks perform several functions: they elect leaders, add transactions into the ledger, and vote for their ancestor blocks via parent link relationships. Prism separates these roles by using three separate types of blocks: proposer blocks for leader election, transaction blocks for transaction inclusion, and voter blocks to confirm transactions. Whenever a block is mined, it is randomly sortitioned into one of the three types of blocks and, if it is a voter block, further sortitioned into one of m voter chains ($m = 1000$ is suggested). This sortition process ensures that miners are unable to choose which type of block they mine. Rather, they simultaneously mine for each chain and only learn the type of the mined block after a valid proof of work is found. Proposer blocks contain a list of references to transaction blocks, as well as a single reference to a parent proposer block. As with Nakamoto Consensus, honest miners will mine on top of the longest proposer chain they are aware of. However, it is the voter chains that determine the final sequencing of proposer blocks (and thus elected leaders).

Define the *level* of a proposer block as its distance from the genesis block of the proposer chain and the *height* of the proposer chain as the maximum level containing any proposer blocks. Voter blocks include a reference to a proposer block in order to cast a vote on it that is subject to two requirements: 1) the voter block is in the longest chain of its respective voter tree, and 2) each voter chain votes for exactly one proposer block at each level. The leader sequence, then, is the proposer block at each level with the highest number of votes among all proposer blocks at that level with ties broken by the smallest hash of the proposer blocks. With the proposer blocks ordered, their references to transaction blocks create an agreed-upon ordering of transaction blocks and thus transactions. The ordered list of transactions must then be sanitized in order to remove invalid or duplicate transactions. See Figure 30 for Prism's structure.

Ultimately, the security of the system is provided by the voter trees that give confirmations/votes to the proposer blocks. Changing an elected leader requires reversing a sufficient number of voter blocks, and each vote is secured by following the longest chain rule

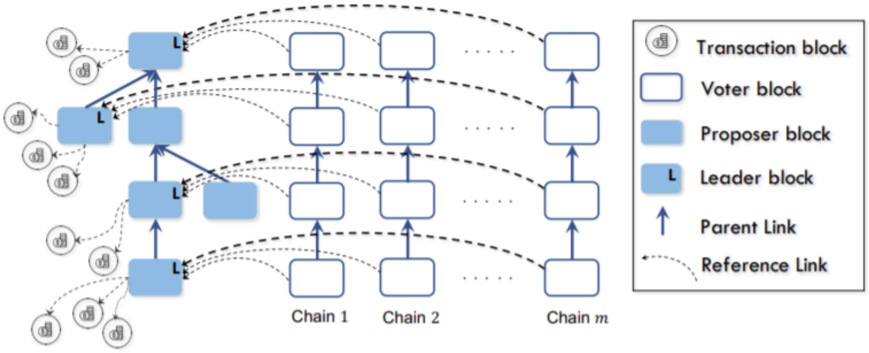


Fig. 30. Prism structure. [276]

in its voter tree. Due to the large number of voter chains and the global hash rate being split among them, it is possible to minimize the amount of forking in each voter chain. So long as the voter chains are secure, the proposer chain will be secure as well. As with the more abstract parallel chain protocol above, the existence of many voter chains substantially improves the latency of the transaction settlement.

Prism has security proofs in synchronous networks in a couple of different models [277, 278], and it can be made capable of handling smart contracts [279]. On the other hand, the protocol as described does not have a clear way of supporting light clients, which would be especially important for a protocol designed to maximize throughput. More research must also be performed to develop a suitable incentive system for Prism.

11.5. Proof-of-Work DAGs

One of the more common design decisions in the distributed ledger space is to utilize DAGs rather than singularly linked blockchains. The primary motivation – as with most of the protocols described above – is to increase the transaction throughput of the system by reducing the negative security ramifications of block propagation latency. DAG-based ledgers are similar to GHOST (Section 11.2) in that blocks incorporate references to more than one previous block but go further by including those blocks directly into the ledger rather than merely using them as input to determine a particular chain to follow. While not a perfect analogy, each block produced acts as a confirmation for all prior blocks it references rather than just one. As a result of this architectural change, blocks can be produced dramatically more frequently, which allows miners to maintain eventual consistency with incomplete information about the state of the DAG.

On the other hand, it also complicates the design and analysis of these systems. For example, [280] investigates the fairness (i.e., whether a miner's rewards are proportional to their hash power) and efficiency (i.e., the fraction of transactions broadcast to the network that are included in the ledger after a certain period of time) of some DAG-based protocols. They found that – unlike Nakamoto Consensus – several DAG-based protocols lack fairness even when all miners are honest because fairness is inherently limited by the number of block pointers that can be included in a given block. Both fairness and efficiency may break down when miners have varying connectivity to the network, and perversely, large miners may benefit from having reduced connectivity. While DAG-based schemes are promising, extreme care must be taken to ensure that they can remain secure and incentive-compatible in the real world.

11.5.1. Inclusive Blockchains and Conflux

One of the early proposals for a DAG-based ledger was the Inclusive protocol [281], a variant of which was later employed in the Conflux system [282, 283]. Inclusive was designed to tolerate larger and more frequent blocks to enable higher throughput without penalizing miners who are poorly connected to the network. The protocol is called Inclusive because it includes transactions from all blocks in the DAG into the final ledger. Unlike some other DAG protocols, Inclusive creates only a single main chain, but that chain incorporates transactions from blocks that are not ultimately accepted into the main chain. In an Inclusive protocol, miners include a reference to every chain tip that they are aware of in their own blocks rather than just one. The DAG formed by these blocks and references allows the "simulation" of any chain selection rule, including the longest chain rule and GHOST. The first listed reference should be for the block that would be the preferred chain tip based on the system's chain selection rule.

The algorithm performs a postorder traversal of the block DAG while incorporating valid transactions from off the main chain into the linearized ledger of accepted transactions. For a canonical chain $C = B_1, B_2, ..., B_L$ of blocks B_i, the Inclusive rule will place all the blocks of the system in a particular order, including those outside of C. Let $past(B)$ be the set of blocks reachable from B in the DAG. The ordering operates as follows: for $B_i \in C$, insert before B_i all blocks in the set $past(B_i) \setminus past(B_{i-1})$. The included set of blocks are sorted topologically with ties broken in favor of the lowest block hash. When blocks are ordered this way, some invalid or duplicate transactions will exist, which are then removed from the ledger of transactions.

Fees are given to the miner of the block that a transaction is included in, even if the block is not in the canonical chain. Say a miner mines a block B, and let $T(B)$ be the set of transactions included in the ledger from block B. Depending on the block's location in the DAG (specifically, how quickly the block is referenced by a canonical chain block), the block producer may only get a fraction of the total fees included in $T(B)$.

Let $before(B)$ be the latest block from the main chain that is reachable from B and $after(B)$ the earliest block from the main chain from which B can be reached. If $after(B)$ does not exist, it is considered a "virtual block" with height infinity, representing the location of the next block that a miner with the same view of the ledger would produce. When B is in the canonical chain, $before(B) = after(B) = B$. Let $gap(B) := after(B).height - before(B).height$ describe the delay in a block's publication with respect to the canonical chain. The fraction of the fees collected by the miner of B will be a weakly decreasing function of $gap(B)$. Main chain blocks, as well as blocks that are relatively synchronized with the main chain, receive the full reward. As $gap(B)$ increases, the fraction of the fees that the miner collects decreases until a certain cutoff point where they no longer receive a reward. For example, the reward function may work as follows, where $fee(tx)$ is the transaction fee for transaction tx:

- If $0 \leq gap(B) \leq 3$, the miner collects the full reward, $\Sigma_{tx \in T(B)} fee(tx)$.
- If $3 < gap(B) < 10$, the miner collects $\frac{10-gap(B)}{7} * \Sigma_{tx \in T(B)} fee(tx)$.
- If $gap(B) \geq 10$, the miner gets nothing.

This type of fractional fee scheme has security trade-offs. Providing fees for off-chain blocks is what allows poorly connected and smaller miners to continue receiving significant enough rewards instead of being competed out of existence by larger miners. However, to the extent that miners of off-chain blocks receive rewards, malicious behaviors are encouraged. An attacker who tries and fails to double-spend still receives some revenue from their off-chain blocks, and thus the attack is subsidized. As a result, double-spending is easier in Inclusive than it is in, say, Nakamoto Consensus and thus requires waiting for more confirmations for the same level of security. In addition, Inclusive provides no defense against selfish mining.

The authors of [281] argue that under a number of game theoretic models, miners will attempt to include transactions that minimize collisions with other blocks rather than merely include those with the highest fees. The higher performance of Inclusive stems from this collision-avoiding transaction selection policy because collisions waste bandwidth and other resources. Unfortunately, this may make miners less likely to broadcast transactions with high fees to the network because they would prefer to keep those fees to themselves without risking collisions.

The Conflux protocol is distinct from Inclusive but highly related [282, 283]. The primary difference is that Conflux blocks include two distinct types of references to other chain tips instead of treating all references equally, as Inclusive does. By distinguishing between *parent edges* and *reference edges*, Conflux's safety follows more directly from the safety of the GHOST fork-choice rule that it employs (see Section 11.2 to review this rule). In Conflux, the parent edges act as votes on the proper chain history, whereas reference edges demonstrate only that the referenced blocks were produced before the block that referenced them. When a miner mines a block, it sets the parent edge to be the chain tip that follows GHOST, and the sequence of parent edges create a canonical chain called the *pivot chain*. In other words, miners compute the pivot chain based on GHOST, then set the tip of the pivot chain as their parent edge, and any other chain tip off of the pivot chain is set as a reference edge. Each pivot chain block establishes a new *epoch*, where an epoch contains every block in the DAG that is reachable from the pivot chain block and is not in a prior

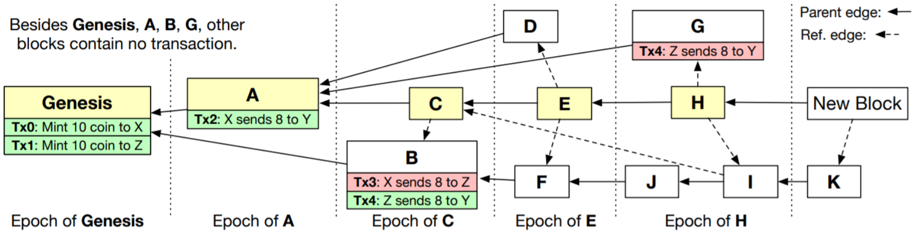


Fig. 31. Example of local Conflux DAG state. The pivot chain is composed of the yellow blocks, which are used to partition the DAG into epochs. The block total order is: Genesis, A, B, C, D, F, E, G, J, I, H, and K. The transaction total order is: Tx_0, Tx_1, Tx_2, Tx_4, with Tx_3 and a duplicate Tx_4 excluded. [282]

epoch.

Conflux then totally orders the blocks and uses the block ordering to totally order the transactions. To totally order the blocks, the blocks are first grouped together by epoch – all blocks in an epoch come before any block in the following epoch. Within each epoch, blocks are ordered based on a topological ordering that follows the reference edges, breaking ties in some deterministic way (e.g., the smallest hash). The total ordering of transactions follows logically from the ordering of the blocks, with invalid and duplicate transactions removed. Figure 31 shows an example of Conflux's procedure for creating a total ordering of transactions from the block DAG.

Conflux includes an additional rule that is intended to mitigate the threat of certain liveness attacks against GHOST [283, 284]. The structure of the past-subgraph of the DAG is used to detect when an attack is underway, at which point the fork-choice rule is adjusted to assign blocks an adaptive weight: the weight is h with probability $\frac{1}{h}$, or zero otherwise. When no attack is detected, the weight is one. The adaptive weight should help miners converge on a single chain by disrupting the ability of an adversary to balance the weights of each chain against each other.

11.5.2. SPECTRE and Phantom

SPECTRE (or "Serialization of Proof-of-work Events: Confirming Transactions via Recursive Elections") is a DAG-based algorithm that was designed to allow for very high block creation rates without reducing the honest majority consensus bound (i.e., regardless of network conditions, an attacker requires a majority of the computational power to prevent agreement) [285]. Unlike the proof-of-work algorithms considered so far, SPECTRE does not lead to a total ordering of transactions but rather a partial order, so it cannot be used for arbitrary smart contracts. The algorithm outputs a pairwise ordering between blocks, so it is possible that cycles may exist in the ordering. If a total ordering is required, the Phantom protocol, which was inspired by SPECTRE, should be used instead [286, 287]. However, SPECTRE's performance stems largely from not running a complete consensus algorithm.

As a result, miners need not be concerned with how well-synchronized other miners are. This allows very low latencies for transaction acceptance. As block creation rates increase, this latency reduces approximately to the propagation delay for reaching a large amount of honest nodes. To understand how these algorithms operate, a few definitions are needed:

- $past(B,G)$ represents blocks that were provably created before block B in the DAG G. Once block B is mined, the set $past(B,G)$ does not change.
- $future(B,G)$ represents blocks that were provably created after block B in the DAG G.
- $cone(B,G)$ is the set of blocks in the DAG G that have been ordered with respect to B. That is, $cone(B,G) := past(B,G) \cup future(B,G) \cup \{B\}$.
- $anticone(B,G)$ is the set of blocks in the DAG G that are not directly ordered compared to B. That is, $anticone(B,G) := G \setminus past(B,G) \cup future(B,G) \cup \{B\}$.
- $tips(G)$ is the set of chain tips, or the blocks without any incoming edges in G.
- $virtual(G)$ is a non-existent, hypothetical block that satisfies $past(virtual(G)) = G$. It represents the next block that a miner would create if their view of the DAG was G.

As with other DAG protocols, miners include references to all known chain tips inside their blocks. The block DAG is then interpreted in order to extract a partial ordering of transactions that everyone can agree on. Naturally, if a block $X \in past(Y)$, then X precedes Y, or $X \prec Y$. Similarly, if $tx_1 \in X$ and $tx_2 \in Y$, $tx_1 \prec tx_2$. For a pair of blocks $(X,Y) \in G$, all other blocks $B \in G$ are interpreted as votes on the pairwise ordering of X and Y. The voting rules of SPECTRE correspond to a generalization of Nakamoto Consensus's longest chain rule applied to DAGs. Specifically, for a block $B \in G \cup virtual(G)$, voting uses the following rules (an example of these voting rules being applied to a DAG can be seen in Figure 32):

1. If $B \in future(X)$ but $B \notin future(Y)$, then B's vote is that $X \prec Y$. This rule gives votes to blocks that were published quickly instead of being withheld.
2. If $B \in future(X) \cap future(Y)$, then B's vote is the same vote as $virtual(past(B))$. Ties are broken arbitrarily in some agreed upon way. This rule, as well as rule 4, gives more votes to blocks that are already supported by the majority in order to help nodes quickly converge on the same precedence relations.
3. If $B \notin future(X) \cup future(Y)$, then B's vote will match that of the majority of blocks in $future(B)$. This rule counters pre-mining attacks where a block is withheld for a long time.
4. If $B = virtual(G)$, then B's vote will match that of the majority of blocks in G.

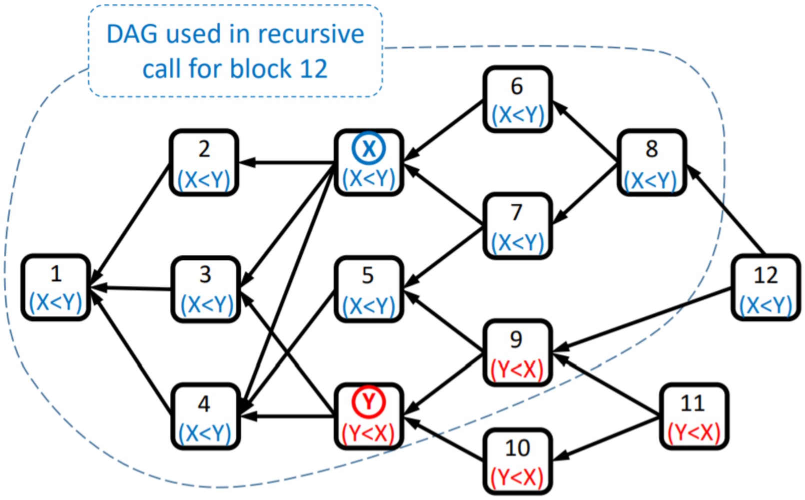


Fig. 32. SPECTRE voting example. Blocks X and 6-8 vote $X \prec Y$ because they see X but not Y in their $past()$. Blocks Y and 9-11 vote $Y \prec X$ for the same reason. Blocks 6-11 vote based on rule 1 in the text, and blocks X and Y vote using rule 5. Blocks 1-5 vote $X \prec Y$ because they see more $X \prec Y$ than $Y \prec X$ votes in their $future()$. Blocks 1-5 use rule 3 to determine their votes. Because block 12 is in $future(X) \cap future(Y)$, it votes according to a recursive call on the DAG that does not include blocks 10, 11, or 12, which are not in its $past()$. Block 12 votes using rule 2. Finally, a miner with this view of the DAG would create a block that references blocks 11 and 12, and its vote would be that $X \prec Y$ based on rule 4. [285]

5. If $B \in \{X,Y\}$, then B's vote is such that for any block $Z \in past(B)$, $Z \prec B$, and for any block $Z' \notin past(B)$, $B \prec Z'$.

For a merchant to consider a transaction tx as confirmed, all of the transaction's inputs must be confirmed. Two additional conditions must hold, assuming tx is contained in a block B. If there are conflicting transactions in $anticone(B)$, then the blocks where those conflicting transactions reside must be preceded by B. Finally, any conflicting transactions in $past(B)$ must have been rejected.

The voting procedure described above is fairly unintuitive, but a few key ideas can help show why SPECTRE is secure. First, if a block is seen by honest miners, then those miners will (directly or indirectly) reference it, such that it ends up in the past sets of newly created honest blocks. By rule 5, blocks support other blocks in their past. This implies that an attacker who withholds their blocks will lose votes. Second, when a block X has the majority of votes compared to a potentially conflicting block Y, this majority quickly becomes amplified, allowing miners to converge and making it challenging for an attacker to reverse the precedence relation. This is because, by rules 2 and 4, new blocks will vote the same way as the majority of blocks in their past. Third, by rule 1, blocks created in the past will vote based on which competing block is in its future. This incentivizes miners to reference recently created blocks in order to solicit those blocks' votes. A block created by an attacker that does not reference recent blocks (perhaps because it was withheld in order to try to double-spend) will lose votes compared to the honest miners. Finally, blocks from the past will counter-balance pre-mining attacks where blocks are withheld. By rule 3, blocks produced by honest miners who are unaware of the pre-mined block will vote in line with the majority of blocks in their future set. To see why this matters, consider an attacker who attempts to double-spend by pre-mining a long secret chain and sending a conflicting transaction to a merchant, which ends up in a publicly known block. Honest blocks produced while the double-spending block was withheld will vote with the majority of blocks in their future set. As long as the attacker does not have the majority of the computational power, it becomes exponentially more likely as time passes that past blocks will support the block paying a merchant.

Unfortunately, SPECTRE has a number of implementation complexities. The handling of difficulty adjustments and transaction fees are out of scope for this document. Furthermore, there are practical limits to how many blocks can be referenced by any given block and, thus, the extent to which block creation rates can be increased before the overhead of the references themselves becomes prohibitive. That said, the throughput and latency improvements are still substantial.

The Phantom protocol is based off of the same ideas as SPECTRE but uses some additional techniques to enforce a total ordering of transactions. Thus, it is suitable for a broader variety of settings at the cost of higher transaction settlement latency [286, 287]. Note that the original Phantom protocol from [286] suffered from a liveness issue where a low-hash-rate attacker could delay transaction confirmation indefinitely, as pointed out in [282]. This is fixed in [287]. Technically, Phantom requires solving an NP-hard problem, and GHOSTDAG is the algorithm that approximates an "ideal" Phantom, but this section will use the more well-known name Phantom for both.

Honest miners in Phantom (as well as other systems) are expected to broadcast their blocks as quickly as they are found. They are also expected to reference as many chain tips as they are aware of and ideally be well-connected to each other. If the majority of miners are honest, this should result in a cluster of blocks that are well-connected to each other. Adversarial miners may, in contrast, withhold blocks temporarily or create new blocks that do not reference publicly visible chain tips in order to overwrite other blocks. These malicious behaviors are detectable based on the structure of the DAG because they will not be as well connected to the majority honest cluster (although, as with Nakamoto Consensus, this malicious behavior cannot be distinguished from a network partition).

In more detail, let Δ be the upper bound on the network's propagation delay, and assume that an honest miner found a block B at time t. Then all blocks that were broadcast by time $t-\Delta$ will be in $past(B)$, and B will be in the past set of any honestly produced blocks after $t+\Delta$. Recall that $anticone(B)$ is the set of blocks that are not referenced by B and that do not reference B. This implies that $anticone(B)$ will contain only malicious blocks (ignoring network partitions) and a small number of honest blocks produced in the 2Δ-sized interval $[t-\Delta, t+\Delta]$. Due to the nature of proof of work, this number of blocks will be bounded by some k with high probability. That is, the parameter k is a function of the network delay. The idea behind Phantom is to recognize and select the largest cluster of blocks in the DAG by observing these anticones. Specifically, Phantom attempts to solve the *maximum k-cluster subDAG problem*: given a DAG $G = (V, E)$, Phantom attempts to output a maximally sized subset $S* \subset V$ such that $|anticone(B) \cap S*| \leq k$ for all $B \in S*$.

Phantom generalizes Nakamoto Consensus to a DAG, where every block produced ultimately ends up in the ledger, but adversarially produced blocks should appear later in the total ordering. This is done by extracting a well-connected cluster of blocks that are presumed to be honest by the honest majority assumption and then totally ordering blocks in a way that prioritizes blocks inside of the cluster. Phantom uses a greedy algorithm that approximates a solution to the maximum k-cluster subDAG problem, where blocks within the cluster are called Blue blocks, and ones outside of the cluster are Red. The algorithm, $OrderDAG(G, k)$, outputs a set of Blue blocks and an ordered list of all blocks in G:

1. If $G = \{genesis\}$, then return $[\{genesis\}, \{genesis\}]$.
2. For $B \in tips(G)$ do: $[BlueBlocks_B, OrderedBlocks_B] \leftarrow OrderDAG(past(B), k)$. That is, use recursion on the chain tips in order to find the best tip.
3. Inherit the Blue set of the best tip, and add this tip to the Blue set and the end of the ordered list.
    - $B_{max} \leftarrow argmax\{|BlueBlocks_B| : B \in tips(G)\}$ (break ties according to lowest hash)
    - $BlueBlocks_G \leftarrow BlueBlocks_{B_{max}}$
    - $OrderedBlocks_G \leftarrow OrderedBlocks_{B_{max}}$
    - add B_{max} to $BlueBlocks_G$
    - add B_{max} to the end of $OrderedBlocks_G$
4. For $B \in anticone(B_{max}, G)$ do (in some topological ordering): If $BlueBlocks_G \cup \{B\}$ is a k-cluster, then add B to $BlueBlocks_G$. Either way, add B to the end of $OrderedBlocks_G$.
5. Return $[BlueBlocks_G, OrderedBlocks_G]$.

To summarize, the DAG is colored recursively, so the Blue set will include all of the Blue blocks from the chain tip with the largest Blue set in its past (denoted B_{max}). New Blue blocks are then added from blocks that lie outside of $past(B_{max})$ but not before checking whether the k-clustering would be violated by their inclusion (step 4). The ordering works similarly and begins by inheriting the ordering from B_{max}. Then B_{max} is next, and then blocks that lie outside of $past(B_{max})$ are topologically ordered in some agreed-upon way. An example of this algorithm is in Figure 33. Once blocks are totally ordered, transac-

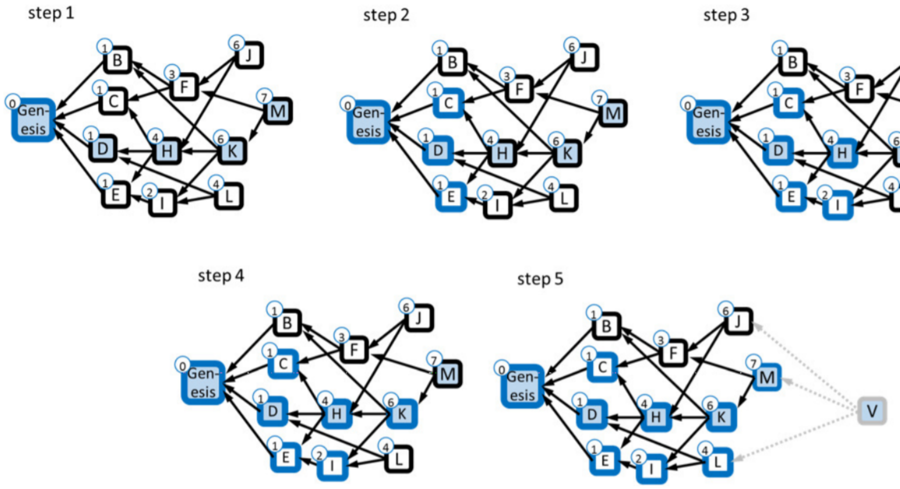


Fig. 33. Phantom example, constructing $BlueBlocks_G$ with parameter $k = 3$. For each block, the circle near it represents its "score," or the number of Blue blocks in its past set. The algorithm begins at the highest scoring tip, $B_{max} = M$, and greedily selects its predecessors: K, H, D (arbitrarily breaking the tie between C, D, and E), and the genesis block. This creates a chain from B_{max} back to genesis. The block V is a hypothetical "virtual" block that references all tips of the DAG. The set $BlueBlocks_G$ begins empty and is constructed recursively as follows. (1) Visit D, and add the genesis block to $BlueBlocks_G$ since it is the only block in $past(D)$. (2) Visit H, and add blocks C, D, and E to $BlueBlocks_G$ because they are in $past(H)$. (3) Visit K, and add blocks H and I to $BlueBlocks_G$. Block B is in $past(K)$, but $anticone(B)$ has 4 Blue blocks and thus is not included. (4) Visit M, and add K to $BlueBlocks_G$. Block F is not added because, like block B, $anticone(F)$ has more than k Blue blocks. (5) Visit the virtual block, V, and add M to $BlueBlocks_G$. Block L is not added because $anticone(L)$ includes C, H, K, and M, which exceeds k (there is an error in the image, and L should not be colored Blue). Block J is not added because the inclusion of J would add another Blue block to to $anticone(I)$, which already contains blocks C, D, and H. [287]

tions are totally ordered in the natural way, removing duplicates and invalid transactions. Note that, similar to Inclusive (Section 11.5.1), miners are incentivized to randomize their transaction selection in order to maximize fees, though this may also encourage them to withhold high-fee transactions.

It is instructive to compare Phantom to other protocols. Recall that Δ is the upper bound on network propagation delay, but as with Nakamoto Consensus, its value is unknown. That said, it is assumed to be smaller than a constant Δ_{max}, which is used to derive the hard-coded parameter k. This parameter is the maximum number of blocks that may not be referenced by each other that can be created by the full mining network over the course of a single delay (and a larger k requires increasing the waiting time for transaction settlement). Phantom's security model differs from that of SPECTRE primarily due to the reliance on Δ_{max}, which must be used explicitly in Phantom but is what allows a total order to be established.

Let the block creation rate of a system be λ ($\lambda = \frac{1}{600}$ blocks per second in Bitcoin, for example). In Nakamoto Consensus, the security threshold of the system decreases toward zero as $\Delta\lambda$ increases. That is, the less time there is to synchronize between blocks being found, the lower the security threshold. In contrast, as long as $\Delta \leq \Delta_{max}$, Phantom's security threshold is at least $\frac{1}{2} * (1 - \varepsilon)$ for some small ε. As a result, Phantom can tolerate much higher block creation rates and throughput while maintaining security under honest majorities. Put differently,

- Nakamoto Consensus assumes that $\Delta\lambda \ll 1$.
- The parallel chains approach with m chains, described in Section 11.4, assumes that $\frac{\Delta\lambda}{m} \ll 1$.
- Phantom assumes $\Delta\lambda \ll k$.

11.5.3. Tangle

The other DAG-based protocols discussed in this section construct DAGs where the vertices are blocks, but some designs – including the Tangle – use transactions as vertices instead [288]. A primary motivation for the Tangle structure is to enable fee-less transactions by allowing typical clients to operate as miners. Instead of submitting transaction fees, each transaction is accompanied by a small proof of work that includes validating and approving two other transactions within the Tangle.

The most important concept in a Tangle-based system is the *tip selection strategy*, or how a client chooses which two transactions at the "tip" of the DAG to reference and approve. This policy cannot be imposed by the network, so it is imperative that an incentive-compatible and secure default exists. The security assumption behind the Tangle is that there must be a large enough inflow of transactions posted to the network that are generated by honest clients to outweigh the computational ability of an adversary. Honest clients must frequently issue transactions for this to happen and for the Tangle to work in a permissionless environment without centralized coordination [289]. If this assumption holds, the Tangle can maintain a partial order (not a total order) over transactions. As with SPECTRE (Section 11.5.2), this makes it unsuitable for generic smart contracts.

Transactions in the Tangle have a *weight* associated with them, which corresponds to the amount of work performed to issue it. The *own weight* of a transaction specifically refers to the work performed to issue a transaction, which is normalized to one in this document. Transactions also have a *cumulative weight*, which is the transaction's own weight plus the sum of the own weights of every transaction that approves of it (directly or indirectly). A transaction's *score* is its own weight plus the sum of the own weights of all transactions directly or indirectly approved by it. Stated differently, a transaction's cumulative weight is the sum of the own weights of the transaction's future set and itself, while the transaction's score corresponds to the own weights of all transactions in its past set and itself. A transaction's *height* is the length of the longest path from it to the genesis transaction, and its *depth* is the length of the longest (reverse) path from it to some chain tip.

The originally proposed tip-selection algorithm is dubbed the *Markov chain Monte Carlo* algorithm (MCMC). It starts by choosing some locations in the Tangle and then performing random walks toward chain tips. Specifically, where H_{tx} is the cumulative weight of a transaction tx in the Tangle, and using W and α as parameters, a client selects chain tips to approve as follows:

1. Consider all transactions in a particular range of the Tangle, $[W, 2W]$, as possible starting locations. Randomly choose N of these transactions.
2. Perform independent, biased random walks from these N locations based on prior approvals, such that the walk moves from tx_1 to tx_2 only if tx_2 directly referenced tx_1. Let the set of transactions that directly reference tx_1 be denoted TX. The probability of transitioning from tx_1 to tx_2 is $\frac{e^{-\alpha(H_{tx_1}-H_{tx_2})}}{\Sigma_{tx \in TX}\ e^{-\alpha(H_{tx_1}-H_{tx_2})}}$.
3. Once two of these random walks end at chain tips, select those tips. However, if a random walk ends up at a chain tip "too quickly," it may be considered a *lazy tip* and discarded. A lazy tip is one that approved of an old transaction in order to avoid the effort of verifying transactions. Lazy tips have a low probability of being selected because the cumulative weight of the lazy tip and any others would be substantial.

The parameter α used in MCMC is important. Higher values of α (closer to one) are more "deterministic" and provide a better defense against various adversarial strategies because they increase the chance of the random walk moving to high-scoring tips. A lower value of α (closer to zero) makes the system more stable with respect to transaction confirmation times. A high α will result in many stale tips that need to be reattached to the Tangle in order to achieve confirmation.

Some potential attacks on the Tangle include *parasite chain* attacks, *large weight* attacks, and *splitting* attacks. The parasite chain attack, displayed in Figure 34, is a classic double-spend attack. The attacker builds a subtangle in secret while occasionally referencing the main Tangle in order to inflate the secret subtangle's score. When the attacker has less computational power than the honest portion of the network, the parasite chain attack is challenging for the same reason that selecting lazy tips is unlikely: the attacker's subtangle is likely to have lower cumulative weight, so the random walks will likely remain on the main Tangle. Other techniques for detecting and mitigating parasite chain attacks have been proposed as well [290].

In a large weight attack, as shown in Figure 35, an adversary attempts to double-spend by putting an especially large amount of work into the double-spending transaction in order to outweigh the honest portion of the Tangle. To protect against this attack, there should be an upper bound on the own weight of any given transaction.

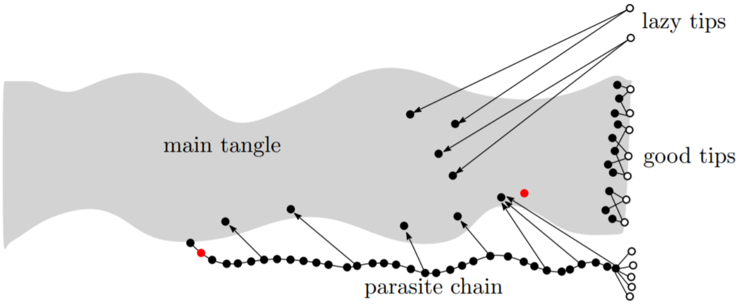


Fig. 34. Parasite chain attack against the Tangle. The two red circles indicate a double-spend attempt by the attacker. [288]

Finally, a splitting attack is one in which the adversary attempts to divide the Tangle into two incompatible branches and then keep them balanced over time until they can spend the same funds on both sides of the split. The attacker issues conflicting transactions near the beginning of the split so that the two sides cannot be reconnected. If honest miners are divided between the two subtangles, a low-hash-rate attacker can attempt to mine on whichever side of the split is required to maintain balance. Starting the random walks at transactions with greater depths in the Tangle makes this attack more challenging to pull off. In addition, the splitting attack is much easier with low α. When higher, random walks quickly converge toward the subtangle with higher cumulative weight.

In addition to the explicit attacks on the Tangle, there are potential game-theoretic issues relating to the incentives of the participants. Intuitively, in order to encourage more clients to approve of one's transactions, one would want to place their transactions in the heavier subtangles because that is where the random walks are more likely to end up. This suggests that incentives are aligned properly, but an alternative strategy would be to simply remember the last 10 or so transactions that were gossiped and approve two of those when issuing a transaction instead of storing the full Tangle.[2] Assuming that most other clients are honest, then the most recently seen transactions are likely to be where random walks end up. If the majority of clients were to use this strategy, then transactions would continue to be approved and everything would appear normal, but the actual security bound

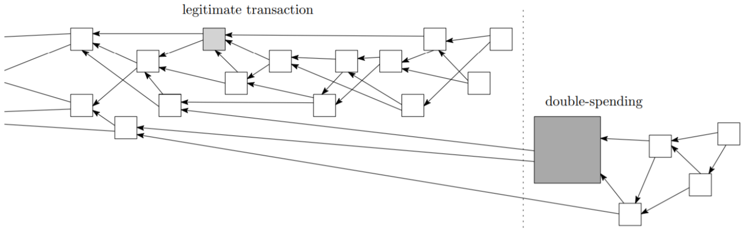


Fig. 35. Large weight attack against the Tangle. [288]

---

[2] Suggested in https://twitter.com/AlexSkidanov/status/1130505820930695169.

decreases toward zero. An attacker can cause a deep fork by quickly posting multiple (in this case, more than 10) transactions in a row on a different subtangle, at which point all of the lazy clients will immediately switch to it and begin approving them.

A number of works have explored the ramifications of alternative tip-selection algorithms [291–294]. Simple algorithms – such as uniformly random tip selection and the use of unbiased random walks – are highly susceptible to parasite chain attacks, while using biased random walks (MCMC) with large α protects against attacks [291]. A modification to the MCMC algorithm that makes parasite chain attacks more difficult while maintaining a lower α is to incorporate the derivative of the cumulative weight with respect to time in the random walk probabilities [292]. Another proposal, dubbed G-Iota, attempts to maintain a higher α while mitigating the issue that this leads to more honest chain tips becoming stale [293]. Finally, the E-Iota proposal attempts to reduce the number of random walks that need to be executed while maintaining the best security guarantees of MCMC and G-Iota [294]. The algorithm is parameterized by $p_1 < p_2 < 1$. Any time the client needs to select new tips, it generates a random number r. If $r < p_1$, the client uses N uniform random walks for tip selection. If $p_1 \leq r < p_2$, it uses N biased random walks with a low α value. If $r \geq p_2$, it uses N biased random walks with high α.

11.5.4. Meshcash

Meshcash is a modular consensus algorithm that combines aspects of proof of work with asynchronous binary Byzantine agreement (ABA) protocols (see Section 6.1.1) [295]. It involves a slower proof-of-work protocol, dubbed the "tortoise," that provides eventual consensus, as well as a quicker "hare" protocol. If the hare protocol succeeds, agreement occurs quickly, but if it fails, the tortoise protocol will ensure eventual consistency. The Meshcash protocol provides the following minimal security guarantees, which are not tight:

- If an adversary controlling less than $\frac{1}{3}$ of the system's computational power is unable to disrupt the hare protocol, then the Meshcash protocol achieves consensus.
- The tortoise protocol achieves consensus against adversaries who control less than $\frac{1}{15}$ of the computational power, regardless of the outcome of the hare protocol.

Both the ABA protocol and the tortoise protocol rely on a *weak common coin* (as defined in Section 2.7). This is implemented via the proof-of-work algorithm with an expected block time of T (e.g., $T = 600$ seconds in Bitcoin). For a player P beginning at time t, the weak common coin protocol works as follows:

1. P waits until time $t + T$ and keeps track of the set of valid blocks received between time t and $t + T$, denoted S_P.
2. P sorts the blocks in S_P by their hashes. The weak common coin output is the least significant bit of the smallest hash block in S_P.

The Meshcash ledger is constructed as a layered DAG, where each block belongs to a particular layer and references blocks from earlier layers (the number of layers that can be referenced by miners depends on the hare protocol). The hare protocol is used for consensus on blocks from more recent layers, while the tortoise protocol orders blocks in the more distant past. Honest miners must remain relatively synchronized with respect to the layers they are participating in, so the layer counter is incremented whenever a miner sees a threshold of valid blocks in a given layer. The hare protocol has two additional requirements:

1. Blocks on layer i are valid only if there are at least T_{min} valid blocks on layer $i-1$ in their past sets. This prevents miners from pre-mining blocks in future layers.
2. For a block B in layer i, honest miners with layer counters in the range $[i+t, i+s]$ agree on the validity of B and will continue to do so throughout the interval $[start_{i+t}, start_{i+s+1})$, where $start_i$ is the time that the first honest miner entered layer i. This property is called *limited* $[t,s]$*-consistency*.

A variety of suitable hare protocols are possible. In the hare protocol, blocks on a given layer are used to elect a committee that runs a traditional ABA protocol (e.g., the one described in Section 6.1.1) off-chain in order to get agreement on the blocks within the layer and then to append signatures to these blocks to attest to their validity. An ABA instance is run for each block in the layer. A block claiming to be in layer i is valid if and only if it has valid signatures from the majority of layer-i committee members.

The tortoise protocol is derived from a hare protocol Π with output interval $[t,s]$. In addition to any requirements from Π, the tortoise protocol requires that honestly produced blocks include references to every chain tip (*view references*) as well as references to all valid blocks in layers $[i-s, i-t]$ (*voting references*). Additionally, given a network propagation delay upper bound of Δ, miners include a *coin bit*, a *before coin bit*, and an *early block bit*. The coin bit is the result of the weak common coin protocol that begins at time $start_i + \Delta$. The before coin bit is set to indicate whether the block was produced before the common coin protocol concluded in order to abstain from voting when the coin bit matters. Finally, the early block bit is set if the block was produced less than Δ time after the layer began so that these blocks abstain from voting if late blocks from the prior layer would make a difference.

The tortoise protocol's block voting procedure, an example of which is in Figure 36, is performed on any block B in layer $i' < i-s$. All valid blocks in the range $[i'+1, i-1]$ vote on B if they were received by $start_i + \Delta$. Votes are weighted by the proof-of-work difficulty of the blocks, and B is valid if this weighted sum is positive. A block Q in layer $j > i'$ votes on block B using the following rules and a protocol-defined threshold θ:

1. When $j < i' + t$, Q votes zero because Q was generated before the hare protocol had a chance to achieve consensus on B.

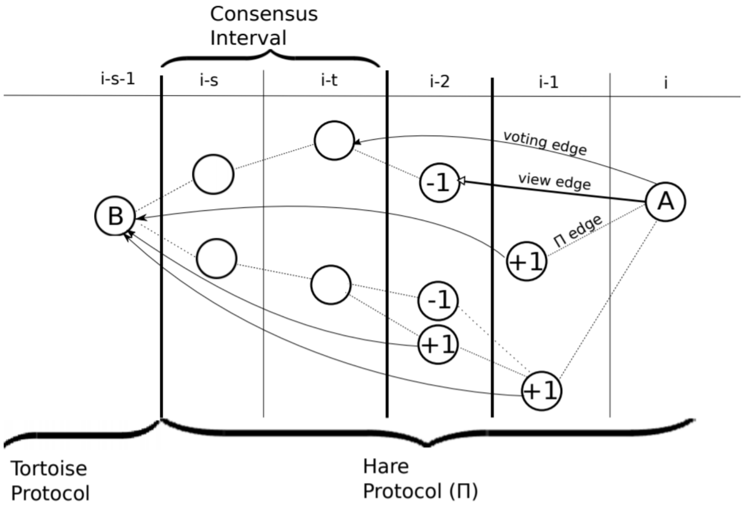


Fig. 36. Example of Meshcash block voting. Here, $s = 4$ and $t = 3$. The blocks in layers $[i-s, i-t]$ vote 0 by rule 1. The blocks in layers $[i-2, i-1]$ vote according to rule 2. Three of these blocks have voting edges to B, and two do not. Block A considers block B valid by rule 3 because the sum of block votes in its past in favor of B is positive. This assumes that, weighted by the proof-of-work difficulty, this sum is greater than θ. [295]

2. When j is in the range $[i' + t, i' + s]$, Q votes 1 if it has a voting reference to B and -1 otherwise.
3. When $j > i' + s$, Q will vote in agreement with the weighted sum of blocks in its past set so long as this sum is outside of the range $[-\theta, \theta]$. However, if this weighted sum is in the range $[-\theta, \theta]$, then Q's vote will be determined by the coin bit and before coin bit:
    - If the before coin bit is set to 1, then Q votes 0.
    - If the before coin bit is set to 0 and the coin bit is set to 1, then Q's vote is 1.
    - If the before coin bit is set to 0 and the coin bit is set to 0, then Q's vote is -1.

This consensus protocol is compatible with any way of allocating rewards, but the authors suggest that transaction fees be distributed to all miners who created blocks in recent layers in order to decrease the incentive for miners to keep high-fee transactions secret. This would also make Meshcash more resistant to selfish mining, not unlike FruitChains (see discussion in Section 11.3). Specifically, the total fees from layer i would be split among the miners of the prior k blocks proportionally to the number of blocks in each layer. By splitting the reward in this way, double-spending attacks become subsidized, but censorship becomes more challenging.

11.6. Proof of Work for Committee Selection

An alternative approach to permissionless consensus is to use proof of work as a Sybil-resistance measure for electing a committee of miners to act as replicas in a standard permissioned BFT protocol. Some of the best advantages of proof of work are removed when using schemes that elect committees in this way. In particular, this introduces the *posterior corruption* issue that separates proof-of-stake protocols from many proof-of-work ones, making the protocols *weakly subjective* (see Section 12.1.2). Essentially, this means that a new node joining the network (or returning online after an extended absence) must acquire a copy of the chain from a trusted party. If past committees are compromised, they can create alternative histories that would be believed by these newly online nodes. In addition, these protocols are less safe against bribery attacks because once a node is in the committee, no additional proof-of-work resource expenditure is needed for it to issue malicious statements.

On the other hand, these protocols are able to provide *responsiveness*, or the ability to confirm transactions at the speed of the actual network delay, rather than the worst-case delay (see Section 5.2). For a permissionless proof-of-work system to be responsive, [296] proved that four conditions must hold:

1. The protocol must know an upper bound on the network delay, Δ.
2. There must be a non-responsive "warmup" period, after which transaction confirmation can become responsive.
3. There must be some "stickiness" to honest nodes. That is, it must take some time for an adversary to corrupt a node or knock it offline. These protocols can only be secure against mildly adaptive adversaries, not fully adaptive ones.
4. Fewer than $\frac{1}{3}$ of the nodes can be corrupt.

11.6.1. Hybrid Consensus

The Hybrid Consensus algorithm uses proof of work to agree on a rotating committee, and the committee uses a traditional BFT algorithm to select transactions [296]. The protocol utilizes an underlying blockchain protocol, such as Nakamoto Consensus or FruitChains, though FruitChains is preferable (see Section 11.3 for discussion on FruitChains). Miners who have successfully mined blocks in the recent past are elected to a committee that then runs the classic PBFT algorithm among themselves (PBFT is described in Section 4).

In more detail, Hybrid Consensus works as follows, where λ is the common prefix consistency parameter (all nodes agree on the contents of the blockchain, except for possibly the trailing λ blocks). A new committee is elected whenever the underlying blockchain adds an additional λ blocks. To elect the committee safely, the trailing $\Theta(\lambda)$ blocks must not be considered because they are potentially unstable. The most recent λ blocks' miners within the common prefix are the new committee members. This means that the same miner may occupy multiple spots in the committee proportional to the number of blocks they have mined in the stable portion of the chain. Prior works that attempted to elect BFT committees using proof of work failed to remove the trailing $\Theta(\lambda)$ blocks, which made them insecure because there was no agreement on the members of the committee itself. By the chain growth property, new committees are elected at regular, somewhat predictable intervals.

Finally, consider chain quality. If the underlying blockchain has at least $\frac{2}{3}$-chain quality, then it can guarantee that for each period of λ consecutive blocks, at least $\frac{2}{3}$ of them will be mined by honest replicas. This is necessary to ensure that at least $\frac{2\lambda}{3}$ of the BFT committee members are honest. If Nakamoto Consensus were used for the underlying blockchain protocol, then $\frac{3}{4}$ of the hash rate would need to be honest in order to achieve $\frac{2}{3}$-chain quality (see discussion on selfish mining in Section 9.4). This motivates the use of FruitChains as the underlying ledger because it has a higher chain quality than Nakamoto Consensus and, thus, allows near optimal resilience.

As described, if an old committee ever surpasses the adversarial corruption threshold, Hybrid Consensus is subject to the issue of posterior corruptions, where new nodes will be unable to tell which of two conflicting chain forks is the correct one. To protect against this, whenever a committee is switched out and a new committee is elected, at least $\frac{1}{3}$ of the old committee will sign a hash of the ledger (or rather, the portion of the ledger that the committee worked on) and include those signatures in the blockchain. This prevents old committees from equivocating.

Transitioning smoothly between multiple consensus committees is also non-trivial. The old committee must undergo some stopping procedure that will overlap with the new committee's reign. Because of this overlap, the new committee's output should be deferred until the old committee has fully completed its term. The stopping procedure involves sending special signed stop instructions until more than $\frac{1}{3}$ of the committee have done so, at which point transactions are ignored.

11.6.2. Solida

Solida is another proposal that combines aspects of proof of work with classical BFT consensus [297]. Unlike with Hybrid Consensus, Solida does not use proof of work to establish an underlying blockchain but rather as a more generic Sybil-resistance mechanism for leader election. A modified version of PBFT (Section 4) is used to commit blocks of transactions or *reconfiguration events* for the consensus committee into the ledger. In order to join the committee, miners must find a proof of work for a computational puzzle, and the existing committee tries to commit the miner's public key, the proof of work, and the system state into the ledger. The miner then joins the committee, pushing the oldest member of the committee out of it.

The normal PBFT leader cannot be in charge of these reconfiguration events because a

Byzantine leader can stall the network until the adversary can generate more proofs of work and become over-represented on the committee. Instead, it is the successful miner who leads the attempt to elect themselves onto the committee. The new member immediately becomes the new PBFT leader for committing transactions. The challenge with this approach is how to address contention issues – that is, when more than one miner finds a proof of work at around the same time. Solida addresses this with a ranking scheme, such that only higher ranked miners can interrupt the reconfiguration of lower ranked miners.

Solida is essentially composed of three subprotocols: the steady state protocol to commit transactions into *slots*, denoted s; the view change protocol to replace faulty leaders; and the reconfiguration protocol. Any given configuration c can have multiple *lifespans*, denoted e, and each lifespan can have multiple views v. Each Solida leader $L(c,e,v)$ is ranked in c, e, v order and uses the following rules:

1. When reconfiguration events are committed, each committee member switches to a new configuration by incrementing c and setting $e = v = 0$.
2. When a new proof of work is found under the current configuration, each committee member increments e and sets $v = 0$.
3. If a timeout occurs and a faulty leader is detected, each committee member increments v. For some hash function H in a system with n committee members, let $l = H(c,e) + v \; mod \; n$. Then $L(c,e,v)$ is the l-th member of the existing committee.

The steady state protocol for committing blocks of transactions is nearly identical to PBFT with two changes: 1) the original pre-prepare, prepare, and commit phase messages include extra contextual information, particularly the (c,e,v) tuple; and 2) an additional "notify" step occurs at the end. After committing a block, replicas send a NOTIFY message that includes a commit certificate and move to slot $s+1$. Upon receiving a NOTIFY message, replicas commit, broadcast their own NOTIFY message, and move to slot $s+1$. The view change protocol requires more substantial adjustments and works as follows when the network delay upper bound is Δ:

1. Upon advancing to the next slot s, replicas set a timer and initiate a view change if it reaches 4Δ before the replica has committed anything in slot s. It does so by broadcasting a VIEW-CHANGE(c,e,v) message to the rest of the committee. If a replica sees $2f+1$ matching VIEW-CHANGE(c,e,v) messages and has not already advanced to a higher view, it forwards the set of VIEW-CHANGE(c,e,v) messages to the new leader $L(c,e,v+1)$. The replica then listens for a NEW-VIEW message from the new leader, and if they do not receive it within 2Δ time, they broadcast a VIEW-CHANGE$(c,e,v+1)$ message.
2. When $L(c,e,v+1)$ receives $2f+1$ matching VIEW-CHANGE messages, they broadcast a NEW-VIEW$(c,e,v+1)$ message (which includes the set of VIEW-CHANGE messages) and enter $(c,e,v+1)$. When replicas receive this message and are not

already in a view higher than $(c,e,v+1)$, they begin the new view and start another timer. If the timer hits 8Δ and no slot has been committed, replicas give up on the current leader and broadcast another VIEW-CHANGE message.

3. When entering view (c,e,v), a replica sends a STATUS message to $L(c,e,v)$ that contains the slot number $s-1$, the value committed in that slot (denoted h), the corresponding commit certificate C, the value accepted in slot s (denoted h'), and the corresponding prepare certificate P. When the leader receives $2f+1$ STATUS messages, they concatenate them into a status certificate, S. The leader then finds the STATUS message that corresponds to the highest last-committed slot $s*$, breaking ties using the message that contains the highest ranked prepared value in slot $s*+1$. Denote the commit certificate and prepare certificate from this message as $C*$ and $P*$, respectively.
4. The new leader broadcasts a REPROPOSE message that includes $s*+1$, h', S, $C*$, and $P*$. Receiving $C*$ allows replicas to commit slot $s*$. In addition, the inclusion of S and $P*$ prove that h' is a safe value for slot $s*+1$, so h' is reproposed for that slot. If the message is valid, replicas commit slot $s*$ if they have not already and then move into the prepare phase of the steady state protocol for slot $s*+1$.

The reconfiguration protocol requires miners to submit a proof-of-work solution to a configuration-specific puzzle, $puzzle(c)$. The puzzle difficulty is periodically updated in order to maintain an expected average reconfiguration interval that is analogous to the block interval in other systems. Naively, this would allow an adversary to gain an advantage over the honest portion of the network by withholding their puzzle solutions analogously to selfish mining. To bound the advantage that withholding provides an attacker, $puzzle(c+1)$ includes any set of $f+1$ commit certificates from the NOTIFY messages in the most recent reconfiguration. This ensures that adversaries learn a puzzle at most 2Δ earlier than honest miners. The reconfiguration protocol operates as follows:

1. A successful miner broadcasts their puzzle solution to the committee members. Committee members then enter view $(c,e+1,0)$ with the miner as the new leader and start a timer. If no slot is committed by the time the timer hits 8Δ, replicas initiate a new view change.
2. In the new lifespan, replicas perform the same actions that they would in the STATUS step of the view change protocol.
3. Let $h*$ be the committed value in the highest committed slot $s*$ in the status certificate and h' be the highest ranked prepared value in slot $s*+1$. The new external leader will then take action based on whether $h*$ and h' are blocks of transactions or reconfiguration events:
    (a) If $h*$ is a reconfiguration event that advances to configuration $c+1$, the leader broadcasts the commit certificate C and gives up joining the committee. An

alternative leader has already completed reconfiguration, so the miner begins working on $puzzle(c+1)$.

(b) If $h*$ is a block of transactions and h' is a reconfiguration event to $c+1$, then the leader broadcasts a REPROPOSE message that includes $(c,e,v=0,s*+1,h',S,C*,P*)$ and terminates the reconfiguration protocol.

(c) If $h*$ is a block of transactions and h' is empty, then the leader attempts to move to slot $s*+1$ by broadcasting a REPROPOSE message that includes $(c,e,v=0,s*+1,h,S,C*,P*)$, where h is a reconfiguration event that would allow the leader to join the committee. Should h be committed, then in slot $s*+2$, replicas advance to the next configuration that includes the miner as a replacement for the oldest committee member.

(d) If $h*$ and h' are both blocks of transactions, then the leader broadcasts a REPROPOSE message that includes $(c,e,v=0,s*+1,h',S,C*,P*)$ in order to repropose h' for slot $s*+1$. Next, the leader tries to join the committee by getting a reconfiguration event, h, into slot $s*+2$. They do this with a PROPOSE message that includes $(c,e,v=0,s*+2,h)$. Should h be committed, then in slot $s*+3$, replicas advance to the next configuration that includes the miner as a replacement for the oldest committee member.

Except for case 3a, committee members handle receipt of the REPROPOSE message identically to how they would during the view change protocol.

This approach differs markedly from the Hybrid Consensus protocol described earlier. One benefit of this approach is that the identities of committee members need not be buried under several blocks' worth of work and, thus, are publicly exposed for a shorter period of time (though they are still exposed, so Solida cannot be secure against fully adaptive adversaries either). Note that as the adversarial hash rate approaches 33%, the number of committee members required to maintain safety blows up toward infinity, and the system is unable to recover if a committee ever has $f+1$ Byzantine replicas in it.

12. Proof of Stake: The Basics

Proof of stake is the first major permissionless Sybil-resistance mechanism introduced after proof of work and is, in part, motivated by the desire to reduce the high electricity consumption of proof of work while achieving similar goals. This section introduces the idea and provides the context necessary for understanding the more advanced protocols described in Section 13. First, some of the earliest proof-of-stake systems are discussed and then used to demonstrate security issues that are unique to proof of stake and how they are addressed in more mature protocols. Next, leader election mechanisms for provably secure instantiations of proof of stake are introduced. The leader election process in proof of stake is fundamentally different from that of proof of work, leading to security issues that result from being able to know in advance when a block producer will be elected. Finally, the block reward mechanism is investigated with a focus on how different reward schemes can lead to or avoid wealth concentration and, thus, centralization.

There are two broad styles of proof-of-stake consensus algorithm. Chain-based schemes are modeled after Nakamoto Consensus with a major difference: using proof of work, a block is constructed before it is determined whether the miner has permission to broadcast it, but in proof of stake, permission to broadcast a block is provided prior to the block actually being constructed [298]. BFT-based schemes effectively operate as permissionless generalizations of traditional, permissioned BFT consensus. With some minor adjustments, many of the permissioned BFT algorithms covered earlier in this document can be made to work in a proof-of-stake context. Chain-based schemes have greater fault tolerance since they are secure so long as the majority of the stake is honest. In contrast, BFT-based proof of stake inherits the security threshold of the underlying BFT scheme, which is typically $\frac{1}{3}$. Chain-based schemes maintain availability during network partitions, while BFT-based ones sacrifice availability for consistency. Unlike chain-based schemes, BFT-based proof of stake can result in low latency while finalizing transactions. The specific properties, of course, depend on the design of the proof-of-stake algorithm.

12.1. Early Attempts at Proof of Stake

The first network to use proof of stake was Peercoin, proposed in 2012 [299]. The network retained an element of proof of work in order to distribute the coins fairly, but the fork-choice rule used for consensus replaces total work accrued with total *coin age destroyed*. Coin age is defined as the number of units of currency times the length of time that the units have been held without use. Peercoin blocks include a special transaction where the block producer consumes their coin age by sending funds to themselves. The first input from this transaction is called the *kernel*, and the kernel is hashed and checked against a target value similarly to how this is performed in proof of work. The target is calculated per unit of coin age spent in the kernel such that consuming more coin age proportionally increases the chance of being able to produce a block. As with Nakamoto Consensus, this process can result in multiple blocks being produced at the same time, but instead of choosing which fork to stay on based on the greatest total work, Peercoin validators prefer the fork with the highest coin age destroyed, where every transaction in a block contributes to the consumed coin age.

This system includes protection against some very basic attacks. For instance, to prevent users from moving their stake from one output to another in order to increase their chance of producing a block, the coin age computation requires a minimum age of one month and is considered zero below this. Centralized, developer-signed checkpoints are then periodically issued in order to ensure that every node agrees on transactions older than one month, which is necessary for verifying the kernel. A winning block producer could potentially use their single proof of stake to create many valid blocks for use in a denial-of-service attack. To prevent this, nodes collect all kernels and associated timestamps they have seen and ignore any blocks with the same (kernel, timestamp) tuple as a previously received block.

After Peercoin, a second generation of proof-of-stake networks launched that entirely eschew proof of work, including BlackCoin and NXT. BlackCoin's consensus, PoS v2, is similar to Peercoin but removes coin age and makes several other small changes [300]. The use of coin age has the unfortunate side-effect of encouraging users to stay offline most of the time and only open their wallets to stake once every month or so. Further, when a user has a large amount of coin age built up, they can produce new blocks almost immediately and more easily execute double-spend attacks. Another observation was that of predictability – the input into the hashing algorithm for leader election does not prevent an adversary from precomputing future proofs of stake, which allows them to know in advance when they might produce several blocks in a row which facilitates double-spend attacks (this is discussed in detail in Section 12.2). PoS v2 mitigates this by including a *stake modifier* in the hash calculation that changes relatively frequently and allows for shorter precomputation windows. Finally, BlackCoin considerably restricts the timestamp rules for blocks, such that a node only accepts blocks with timestamps less than 15 seconds ahead of local time, while the timestamp granularity is changed from one second to 16 second intervals. This gives potential block producers less freedom to produce more blocks by trying to hash kernels with different timestamps.

The previous point is important. If a staker has a large degree of freedom in selecting the input to the hash function used in leader election, they can perform a *stake grinding* attack, where they iterate through these potential inputs until they find one that is especially advantageous to them. An attacker can go through the history of the blockchain and – wherever their stake was selected to be the next block producer – modify the next block header or kernel over and over until it finds one that helps elect them as the next block's producer as well. By changing the granularity of timestamps from one second to 16 seconds, BlackCoin reduced the number of grinding attempts by a factor of 16. Later, a set of incremental improvements was made to PoS v2, leading to the creation of PoS v3 [301]. The staking process for PoS v3 works as follows and loops forever:

1. Nodes check their system clocks and set the (potential) block timestamp to the system time modulo 16.
2. The network difficulty is computed, and the local difficulty target is computed by multiplying network difficulty by the number of coins held in a particular UTXO.
3. The node iterates through each UTXO in its wallet. For each UTXO, the node computes a SHA-256 hash of several pieces of data, including the previous block's stake modifier, some data from the UTXO, and the timestamp.
4. These hashes are compared to the local difficulty targets for each UTXO, and if the value of a hash is less than the target, the node can create a new block. The block

hash is then signed by the public key specified in the special staking transaction.

5. If no hash satisfies the target, then the node waits 16 seconds and tries again with the new timestamp.

Unlike BlackCoin and Peercoin, NXT uses an account model instead of UTXOs [302]. The entire NXT supply was distributed up front in a presale and then encoded into the genesis block. That is, there is no inflationary block subsidy, and block producer rewards consist solely of transaction fees. The NXT leader election protocol is called "forging" (instead of mining) and is described below.

As with Peercoin and BlackCoin, potential block producers hash some input in the hopes that the hash is less than a particular target value, which changes from block to block in a manner analogous to proof-of-work difficulty adjustment algorithms (Section 9.2) in order to maintain 60-second block intervals on average. This target is computed individually for each account and determined in part by a *base target* value common to all users. Let S be the average block interval for the last three blocks, T_p the base target for the previous block, and T_b the base target being calculated for the current block. Then,

$$T_b = \begin{cases} \frac{T_p * min(S,67)}{60} & S > 60 \\ T_p - \frac{T_p * 0.64 * (60 - max(S,53))}{60} & S \leq 60 \end{cases}$$

Here, the constants 67 and 53 bound the size of target adjustments, while 0.64 is included to allow block intervals to be shortened more rapidly than they can increase, which helps reduce the incidence of extremely lengthy block intervals. Each account can then compute its individual target value, T, as $T = T_b * S_p * B_e$, where S_p is the number of seconds that have passed since the previous block and B_e is the effective balance of the account. Only coins that have been in the account for at least 1440 blocks (one day) count toward the effective balance for staking.

Each block includes a *generation signature*, which is a 32-byte hash output. Stakers will concatenate the previous block's generation signature with their public key and hash it to form the next block's potential generation signature. The first eight bytes of this hash are interpreted as an integer called the *hit*. As with proof of work, if the hit is less than the target, the account holder is authorized to produce a block. Unlike with proof of work, generation signatures make it possible to predict which account will produce the next block with fairly high accuracy. The target value increases with every second passed since the previous block, so even if most accounts are offline, a block will eventually be produced. If multiple blocks are produced around the same time, nodes prefer the fork with the highest cumulative difficulty (difficulty being the inverse of the target value). The current block difficulty, D_{cb}, is calculated as $D_{cb} = D_{pb} + \frac{2^{64}}{T_b}$, where D_{pb} is the previous block's difficulty. In addition, NXT eschews centralized checkpoints and instead just maintains a local rule that nodes will never reorg more than 720 blocks.

Another early proof-of-stake variant, *delegated proof of stake* (DPoS), continues to be widely used despite not being formally studied to the degree that many other protocols have been. Technically, a variety of proof-of-stake protocols, including some chain-based ones, allow nodes to delegate their stake to another party, which allows the delegator to contribute to the network's security even while they are offline. Here, DPoS is used to refer to a specific type of protocol where only elected delegates participate directly in consensus as block producers, but all users have the ability to delegate their stake in order to elect a committee of delegates. The first DPoS system, BitShares, went live in July 2014 [303]. Since then, it has been used in a variety of projects, including EOS, Lisk, Steemit, and Tron, as well as some sidechains (sidechains are discussed in Section 16.3).

The mechanics of DPoS are relatively simple: users with stake in the system vote for a committee of n delegates who execute a permissioned BFT algorithm among themselves. Stake delegation usually involves signing a message from the public key that holds the stake, which assigns that user's stake to another user. In many systems, stakeholders can delegate their stake to several potential delegates at a time. Users are ordered by the amount of stake assigned to their keys. The top n become the delegates that make up the committee, and more act as backups who are ready to take over should votes change. This delegation occurs on a continuous basis, so the delegates themselves can be swapped in and out at any time based on changes to the stake delegation distribution. The actual consensus algorithm executed by the committee may differ, but one possibility is that leaders rotate in round-robin fashion, where a block is finalized after $\frac{2n}{3}$ delegates have voted for it by building blocks on top of it. Validators follow the longest chain but will not reorg past a finalized block.

By having only a small number of delegates participate directly in consensus, DPoS can maintain high transaction throughput because the few delegates can run expensive, high-performance servers. In addition, it has relatively low latency due to finalization, unlike most proof of work or chain-based proof-of-stake systems. These performance benefits come at the cost of the increased security risks related to centralization. Delegates are likely to be easier to locate and conduct denial-of-service attacks against because they are likely to be recognized, fairly static entities who operate nodes in data centers, possibly with public IP addresses, and the time slots in which they are supposed to produce blocks are publicly known in advance. This also makes it easier for block producers to collude and perform censorship, and if throughput is high and validation is expensive, it can be hard to vote them out and find replacements.

In DPoS systems, there is a distinct risk of having factions or cartels form, resulting in a form of oligopoly among delegates. In most DPoS models, delegates share block rewards with stakeholders who delegate to them. One implication of this is that stakeholders are likely to delegate only to those potential delegates who are highly likely to be elected, which implies that delegates who have already been elected to the committee are more likely to be voted for and remain in place. To maintain their position in power, these delegates may organize into factions that vote for each other and then impose a requirement that stakeholders must delegate to every member of the cartel in order to receive their "kickback" payment. This is exacerbated by low voter turnout combined with highly unequal distributions of stake, resulting in a handful of big players dominating. This occurred in Lisk, where two large factions controlled a combined 85% of the 101 delegates, with one alone controlling more than half of the total delegates [304]. Similarly, Steemit's top delegates are rarely replaced, and the support of only a few large stakeholders is sufficient to keep them in place [305].

Social choice theory suggests that there may be some risks to the use of delegation in proof-of-stake systems. If some potential delegates are malicious but honest stakeholders are unable to tell whether a potential delegate is malicious or not, it is possible for malicious ones to gain disproportionate power in some circumstances [306]. In addition, delegation favors a minority view, which could amplify the power of a malicious minority of stake [307]. On the other hand, there may exist delegation schemes that avoid the over-representation of minority views on a committee and make it as expensive as possible to elect a certain threshold of delegates (say, $\frac{1}{3}$ of the committee) [308]. More research is needed to gain a better understanding of how delegation impacts the security of proof of stake.

12.1.1. Nothing-at-Stake and Costless Simulation

The most obvious and fundamental difference between proof of work and proof of stake with respect to their ability to act as Sybil-resistance mechanisms is the *nothing-at-stake* problem, sometimes called *costless simulation*. In proof of work, a significant amount of computation is required in order to determine who has the right to produce a block, whereas it takes only a handful of hashing operations to elect a leader under proof of stake. As a result, it costs (practically) nothing for a block producer to append a new block simultaneously to as many chains as possible. This costlessness, combined with the reward for creating blocks, implies that block producers may update the ledger at any opportunity, even if the update would perpetuate disagreement. That is, if a proof-of-stake block producer is presented with multiple competing chains, they do not need to commit to one of them as miners must when using proof of work but rather can produce blocks on both chains for free, giving them the best chance of having a block they produced end up in the canonical chain. As a result, validators may struggle to converge on a single chain as everyone builds on every chain they see.

Costless simulation does not present an insurmountable problem for a proof-of-stake system, and there are some who argue that it hardly presents a real problem for consensus at all [309]. In short, the argument states that equivocating block producers undermine the utility of the system, which should lead to a loss in the exchange value of their stake, and thus there are indeed real costs to producing blocks on multiple chains simultaneously. In order to make sure this condition is sufficient to induce an incentive to maintain consensus, [309] recommends that:

1. There should be a minimum stake requirement in order to participate as a block

producer. A small stakeholder only undermines their own wealth to a very small degree by trying to delay consensus through taking advantage of costless simulation, whereas larger stakeholders undermine their own wealth to a larger degree.

2. Block rewards should be sufficiently low. Under proof of stake, stakeholder incentives come from both the value of the initial coin holdings as well as the block reward. By keeping the block reward component small relative to the initial stake, maintaining coin value by honestly participating in consensus becomes more important than short-term rewards from block inclusion. As the block reward decreases, the minimum stake requirements can become looser.

Consider a scenario in which block producers receive no explicit reward for publishing blocks. If the value of the underlying stake is lowered by delaying consensus, then taking advantage of costless simulation indeed imposes a cost with no offsetting reward. If other stakeholders are honestly following the longest chain rule, then a malicious costless simulator undermines their own wealth both by refusing to add a block to the longest chain and by adding blocks to a shorter chain (and not being compensated for doing so). As a result, following the longest chain rule is an equilibrium for stakeholders when there is no block reward. With a block reward, this equilibrium continues to hold so long as the minimum stake is high enough [309].

In practice, the most common solution to nothing-at-stake instability is to require potential stakers to submit a bond that may be *slashed* (i.e., seized and destroyed by the protocol) if malicious behavior is detected. For example, in Ethereum 2.0 (Section 13.2), participating in consensus requires depositing 32 ETH into a smart contract that is capable of verifying that a particular stakeholder signed equivocating blocks and then seizing the stake from them. Another possible solution is the idea of "virtual ASICs," which mimics proof of work in a proof-of-stake context – stakeholders purchase virtual mining machines and power them with virtual electricity, inducing an operational cost for block production [310].

The early schemes described in the previous section did not take steps to address the nothing-at-stake problem, with the possible exception of NXT. NXT has no block subsidy, so as long as transaction fees are sufficiently small, consensus should be stable based on the argument presented in [309]. None of the other protocols imposed a minimum stake requirement or eliminated block rewards, so they are likely susceptible to costless simulation attacks. Delegated proof of stake is especially susceptible to costless simulation since the value of the stake used to elect delegates is not fully owned by the delegates themselves.

For example, these protocols are more likely to fall victim to bribery attacks than their proof-of-work equivalents. Consider a merchant who waits for six confirmations before providing a good to a customer. After six confirmations, a malicious customer could publicly announce the intention to create a fork that would revert those six blocks and offer a bribe to any stakeholders who would sign blocks on the attacker’s branch [311]. Unlike with proof of work, stakeholders who collude with the attacker in this way risk nothing if the attack fails. Proof-of-work miners have real-world costs imposed on them while mining on the attacker's fork, whereas there is nothing at stake for stakeholders (except for causing consensus delay, as described above).

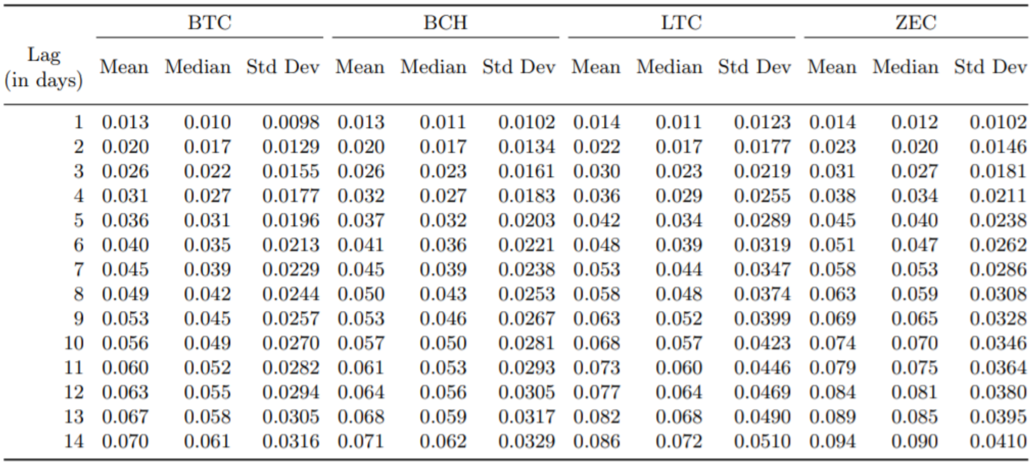
Proof of work also acts as a rate limiter of how quickly consensus-related messages can be generated in a way that proof of stake cannot. A bug shared by at least five proof-of-stake cryptocurrencies using PoS v3 resulted in improper validation of stake and allowed nodes without any stake at all to perform resource exhaustion attacks against any victim node, completely filling RAM or disk space [312].

A concept strongly related to costless simulation is that of *stake shift*. Ideally, leader election would be based on the most up-to-date stake distribution possible in order to provide the maximum amount of Sybil resistance. Unfortunately, this distribution cannot be as recent as desired for two main reasons. First, in chain-based systems such as the ones described above, there is no agreement on the most recent few blocks, so they cannot be used as part of the stake distribution. More generally, even in BFT-based systems where consensus is achieved on a single block before proceeding to the next, the most recent stake distribution cannot be used because the distribution must be fully determined before an adversary can acquire any information on the randomness used to sample from the stake distribution. As a result, for all known proof-of-stake systems, there is a gap between the current stake distribution that establishes the actual incentives for participants and the stake distribution that is in fact used for leader election. This is problematic because a stakeholder can divest themselves of their stake between the time that their stake is used in leader election and the time they are entitled to produce a block. When it comes time to produce the block, they no longer have anything at stake to prevent bad behavior. This suggests that an additional security requirement for proof-of-stake ledgers is that stake cannot change hands "too quickly" [313, 314].

The stake distribution in deployed proof-of-stake protocols typically has a lag of one to 10 days, with chain-based protocols like Snow White and the Ouroboros family being on the longer side, while BFT-based protocols like Algorand have lags on the lower end of that range [315]. The degree of stake shift that occurs during this lag should count directly as part of the adversarial stake when evaluating the security margin of proof-of-stake consensus. More formally, if σ is the stake shift, α is the fraction of adversarial stake, and T is the normal security threshold of the system (i.e., $\frac{1}{2}$ for most chain-based systems and $\frac{1}{3}$ for most BFT-based ones), a proof-of-stake protocol maintains safety if $\alpha < (1-\varepsilon) * T - \sigma$ for some $\varepsilon > 0$. Stake shifts for some more established cryptocurrencies can be seen in Figure 37 and appear to be a few percentage points for typical lags.

12.1.2. Long-Range Attacks, Posterior Corruption, and Weak Subjectivity

Most of the early proof-of-stake protocols described in Section 12.1 included a mechanism for establishing *checkpoints* in the ledger, such that a node will never reorganize the chain to overwrite the checkpointed block. This is due to the possibility of *long-range attacks*,



Lag (in days)	BTC			BCH			LTC			ZEC		
	Mean	Median	Std Dev	Mean	Median	Std Dev	Mean	Median	Std Dev	Mean	Median	Std Dev
1	0.013	0.010	0.0098	0.013	0.011	0.0102	0.014	0.011	0.0123	0.014	0.012	0.0102
2	0.020	0.017	0.0129	0.020	0.017	0.0134	0.022	0.017	0.0177	0.023	0.020	0.0146
3	0.026	0.022	0.0155	0.026	0.023	0.0161	0.030	0.023	0.0219	0.031	0.027	0.0181
4	0.031	0.027	0.0177	0.032	0.027	0.0183	0.036	0.029	0.0255	0.038	0.034	0.0211
5	0.036	0.031	0.0196	0.037	0.032	0.0203	0.042	0.034	0.0289	0.045	0.040	0.0238
6	0.040	0.035	0.0213	0.041	0.036	0.0221	0.048	0.039	0.0319	0.051	0.047	0.0262
7	0.045	0.039	0.0229	0.045	0.039	0.0238	0.053	0.044	0.0347	0.058	0.053	0.0286
8	0.049	0.042	0.0244	0.050	0.043	0.0253	0.058	0.048	0.0374	0.063	0.059	0.0308
9	0.053	0.045	0.0257	0.053	0.046	0.0267	0.063	0.052	0.0399	0.069	0.065	0.0328
10	0.056	0.049	0.0270	0.057	0.050	0.0281	0.068	0.057	0.0423	0.074	0.070	0.0346
11	0.060	0.052	0.0282	0.061	0.053	0.0293	0.073	0.060	0.0446	0.079	0.075	0.0364
12	0.063	0.055	0.0294	0.064	0.056	0.0305	0.077	0.064	0.0469	0.084	0.081	0.0380
13	0.067	0.058	0.0305	0.068	0.059	0.0317	0.082	0.068	0.0490	0.089	0.085	0.0395
14	0.070	0.061	0.0316	0.071	0.062	0.0329	0.086	0.072	0.0510	0.094	0.090	0.0410

Fig. 37. Empirical measurements of stake shift for some high market capitalization cryptocurrencies. While these assets do not use proof of stake, there is little reason to believe that fund movements would differ based on the Sybil-resistance mechanism. Stake shift appears to be smaller on longer-running, more established networks. [315]

where an adversary creates a fork very deep in the chain in an attempt to overwrite the canonical chain. Because producing blocks in a proof-of-stake system is free, a malicious stakeholder suffers no additional costs for attempting to fork the chain from further back in time (even all the way back to the genesis block), as opposed to forking near the chain tip. Checkpoints limit how far back such a reorg can go.

Consider a proof-of-stake system without checkpoints where participating nodes do not have synchronized system clocks. A malicious stakeholder could create a competing blockchain that forks the chain far into the past. If other stakeholders are honest, however, they will not build on the malicious chain unless it is also the longest one, which is unlikely if the attacker has a minority of the stake. However, without synchronized clocks, the adversary can continue to construct blocks that appear to be from the future, ultimately creating the longest chain and succeeding in the attack regardless of the adversarial fraction of stake. As a result, proof-of-stake protocols require synchronized clocks, so most schemes have an external dependency on a clock synchronization protocol, such as the Network Time Protocol (NTP) [316].

Assuming that clocks are synchronized but checkpoints are not in use, such a long-range attack is extremely unlikely to succeed without a majority of the stake backing the attack chain. It is generally assumed that acquiring the majority of the network's stake at any point in time would be prohibitively expensive. However, due to the possibility of *posterior corruption*, this may not be the case. Former stakeholders who have since traded away their stake can collude to extend the ledger from any point at which they did have the majority of stake in the past. This can be rational because it is costless to build an alternate chain and is not detrimental to them because they do not have current stake in the success of the network. Thus, old private keys have some positive value and may be kept after spending the associated stake and then sold to malicious parties. In case of majority posterior corruption, the honest chain can be overwritten from a point deep in the blockchain.

Another variant of long-range attack is *stake bleeding* [317], which exploits the ability of transactions that are valid on the canonical chain to be equally valid on a competing chain. This allows a long-range attacker to use the history of transactions from the canonical ledger on their own attack chain, collect their transaction fees and block rewards, and increase their amount of adversarial stake on the attack chain. As a result, a minority stake attacker can become a majority attacker, particularly if the network has been running for a long time. A fix that prevents stake bleeding – though one with high overhead – is to make transactions context-aware by requiring that they include a recent block hash and are only valid in chains where that block hash exists before the transaction is included.

The most common and simplest fix to implement for each of these long-range attacks is the aforementioned checkpointing. Unfortunately, checkpointing fundamentally changes the system model to one that is *weakly subjective* rather than objective. When a system is objective, a new node (or one that has been offline for an extended period of time) that properly implements the protocol and receives the full set of blocks or other relevant consensus messages will fully agree with the rest of the network regarding the current state. However, a subjective system can have stable states in which nodes do not agree, thus necessitating a social context (e.g., reputation) in order to work. In a weakly subjective system, the new node can come to agree with the rest of the network so long as it has a properly implemented protocol, the complete set of consensus-related messages and blocks, and – crucially – a sufficiently recent (say, N blocks) state that is known to be valid; that is, a checkpoint.

There are two ways in which a node can acquire a sufficiently recent known-valid state: 1) being online frequently (at a minimum, at less than N block intervals) and witnessing the checkpoint block as an active participant of the network or 2) by getting the checkpoint from a trusted party if the node has been offline for more than N blocks (or is first joining the network). As a result, the security model for new nodes and those that have been offline for a while is fundamentally different because they are required to get a checkpoint from a trusted party. Without a trusted checkpoint, these nodes would be unable to differentiate the canonical chain from alternative valid chains. This must be contrasted with (most) proof-of-work schemes that can be objectively validated without introducing a trusted component because total work can be evaluated without having been online.

When a new node attempts to connect to a proof-of-stake blockchain for the first time (or the first time in more than N blocks), it is that user's responsibility to verify a recent state out-of-band by checking a block explorer website, asking businesses they would like to interact with, or asking a friend for a recent block hash. In fairness to proof of stake, there is always a certain degree of subjectivity in terms of downloading a software client from a trusted source because the software must properly implement the protocol for the network that the user wants to connect to. That is, a user who downloads software that falsely claims to implement the Bitcoin protocol can accept a non-canonical, attacker-controlled chain. That said, code signing can help by allowing a user to verify that they are running the desired software for a given network.

In addition to checkpoints, a number of long-range attack mitigations are possible [318]. As mentioned earlier, context-aware transactions can prevent stake bleeding but do not fix posterior corruption. Another mitigation is to use *key-evolving signatures*. Key-evolving signatures allow signatures to be valid only for short time periods while also allowing the private key to evolve as these periods advance despite maintaining the same public key. In the erasure model, where honest nodes are assumed to securely delete their old private keys from local systems, key-evolving signatures can address posterior corruption but not stake bleeding. These kinds of signatures were suggested for use in Ouroboros Praos [319]. Long-range attacks can also be addressed by a change in the fork-choice rule, as is done in Ouroboros Genesis. Roughly speaking, stakeholders using the new rule will prefer the chain that has a greater density of blocks for some period of time after the block where the chains diverge. As discussed in Section 13.1.3, this is not entirely without trade-offs. It may also be possible to protect against long-range attacks using verifiable delay functions, which can be used to prevent an adversary from producing blocks from a time when they were not online but with a significantly reduced security margin [320].

Finally, note that security deposits are not a solution to long-range attacks. While they can address short-term nothing-at-stake issues, users need to be able to eventually withdraw their stake for other uses. Once the security deposit is withdrawn, the ability to hold the stakeholder accountable is eliminated, and they can freely engage in long-range attacks without penalty.

12.1.3. Leader Election, Anonymity, and Security Against Adaptive Adversaries

In the early proof-of-stake protocols described above, two different types of leader election are performed. Most of them use a Bitcoin-esque process of hashing some data, called the kernel, and checking to see if the hash is beneath an agreed-upon target. The kernel structure itself should be designed to reduce or eliminate opportunities for stake grinding. With this style of leader election, offline users do not contribute stake to the security of the system, so the majority of *online* stake must be honest for security to hold. DPoS leader election simply establishes a list of delegates based on stake delegated to them and assigns leaders in round-robin fashion. This is deterministic and results in leaders being known in advance with high probability.

These leader election processes leave much to be desired. Luckily, as proof-of-stake protocols are studied further, the security properties of the leader election subprotocol have become better understood. This section provides a brief overview of some possible improvements in proof-of-stake leader election but is not intended to be comprehensive. More details on a variety of these and other schemes are provided in Section 13.

The first major alternative is a method called *follow the satoshi*, which is used in the Chains of Activity protocol described in Section 13.1.1. In Bitcoin, the smallest possible atomic unit of cryptocurrency is called a *satoshi*. Given a random index into the set of all existing satoshis, one can find the block in which that particular satoshi was minted and then trace the transaction flow from that block to the present to find the public key of whoever currently owns that satoshi. To generate the random seed, each block producer includes a uniformly random bit in their blocks, and at the end of an epoch, the seed is a concatenation of these bits. The seed is used for the follow-the-satoshi elections two epochs later (that is, after skipping an epoch). This procedure results in leaders who are publicly known in advance and, thus, is insecure against adaptive adversaries. To address the problem of offline stake, when an output is selected by follow the satoshi three times without a block being produced, that output can no longer produce blocks until they have spent their coins. This helps maintain consistent block production even when a significant amount of stake may be offline.

Snow White (Section 13.1.2) uses two separate phases to determine each epoch's leaders and the random seed. Each phase lasts for roughly κ blocks, with κ a security parameter. As with Chains of Activity, Snow White block producers include random data in their blocks, which are used in the second phase. In the first phase, the stake distribution used for leader election is determined as the distribution that existed 2κ blocks in the past. The random seed is extracted from the blocks in $chain[-2\kappa : -\kappa]$. As long as a single block is produced by an honest user during this κ-block epoch, the seed can be used securely (this assumption only holds if the majority of the stake is honest). Because the stake distribution is set in stone κ blocks prior to the seed being known, an adversary is unable to adaptively select their public key to perform stake grinding by sending funds to themselves at public keys that would give them an advantage. However, once the adversary knows the next seed used for leader election, they have a one epoch delay in which to try to adaptively corrupt the leaders for the epoch in which the seed will be used. The actual mechanics of leader election are similar to the kernel hash ones above, where a hash of the seed, public key, and timestamp are checked against a difficulty target.

Ouroboros Praos was the first proof-of-stake consensus algorithm to achieve security against fully adaptive adversaries (Section 13.1.3), which it accomplished through the use of a verifiable random function (VRF). The VRF provides the property that only the block producers themselves are aware that they have been elected to produce a block in a given time slot, so other validators – and the adversary – do not know who the leader will be until they have received a signed block with a valid VRF proof from the leader. At that point, it is already too late to perform an adaptive corruption or denial-of-service attack against the leader. Although secure against adaptive adversaries, this method still results in stakeholders knowing locally and in advance when they will have the right to produce a block. This predictability has security ramifications that are discussed in Section 12.2.

In addition to the leader election mechanisms mentioned above and in Section 13, a variety of schemes have been proposed in the literature. Of note is the idea of *single secret leader election* (SSLE), which has some advantages over VRFs [321, 322]. In particular, while VRFs have the benefit of hiding the identity of potential block producers until the blocks are announced, they can result in there being no leaders elected to a given time slot or multiple leaders being elected within the same time slot. In the latter case, multiple leaders can cause undesirable forks in the chain. An SSLE scheme has the following properties:

- *Uniqueness*: Each election results in exactly one leader chosen.
- *Fairness*: If there are N registered users in the system, then each user has a $\frac{1}{N}$ chance of being elected in a given time slot. Further, a single honest user participating in the protocol should suffice to prevent a set of malicious participants from biasing the results of the election.
- *Unpredictability*: If the adversary does not control the elected leader, the adversary cannot learn which user was elected.

Unfortunately, concrete protocols for SSLE tend to rely on very advanced cryptographic primitives, such as indistinguishability obfuscation, threshold fully homomorphic encryption, and public key encryption with keyword search. Another, more practical one is secure under the decisional Diffie-Hellman assumption and uses random shuffles but requires relaxing a security property. SSLE protocols are promising, but more work should be done to improve their performance and make them secure using less exotic cryptography.

The leader election schemes discussed thus far provide neither privacy nor anonymity to the leader. That is, in each case, the leader must reveal their public key to the network in order to create a block. Similarly, the above schemes also provide some information on the quantity of stake owned by the block producer. It is desirable to have leader election protocols that can hide such information, and some work has been done toward this end. For example, in the Ouroboros family of protocols, the Ouroboros Crypsinous proposal is designed so that leader election can occur even in a system where privacy-preserving techniques are used to hide the number of coins associated with each public key, although it does not provide anonymity for the chosen leader and their public key [323].

Other proposals focus on providing anonymity for proof-of-stake leader election [324, 325]. Note how in proof of work, the leader election process does not depend on the identity of the miner because they can include a fresh public key in every block they mine, and the proof is just a valid puzzle solution. Proof of stake, on the other hand, cannot separate the identity of the selected leader from their proof of eligibility. That said, advanced techniques can be used to hide this identity, even if it is used as part of the proof.

The approach in [324] has the ledger store commitments to the total number of coins associated with a public key rather than the amount itself, and stakeholders use NIZK proofs to show that they were elected properly. By itself, this would still leak information based on the frequency with which a given public key is elected and thus provide clues as to the balance of an account. A new *anonymous VRF* (AVRF) primitive addresses this. In an AVRF, multiple verification keys exist for the same private key, and given two proofs for different inputs under different verification keys, an eavesdropper cannot tell whether the proofs were generated using the same private key. The proof statement for the private leader election takes the list of all accounts as an input, proves knowledge of a private key corresponding to a public key in the list, and proves that the stake in that particular account won the election. Unfortunately, the approaches used in both [323] and [324] fail to account for information leaks at the network level and can have their anonymity attacked by an adversary who can control the network delay facing targeted parties. By influencing stakeholders' local views of the network, the adversary can infer stake quantities based on how their targets publish blocks and which views those blocks are consistent with [326].

An alternative approach is described in [325], which proposes an anonymous variant of Algorand's leader election scheme (further detailed in Section 13.4.2). Each party P_i is identified by a public key pk_i. All roles that one could be a leader for are given a tag, tag, that may include things like the protocol round, the specific step or role in the round, and the random seed. P_i checks whether they are eligible for tag by signing tag with their private key sk_i, creating signature σ_i. This signature is input into a hash function H, generating $y = H(\sigma_i)$. If this output is below a threshold T, P_i has been selected, and proves this with (y, σ_i). Verifying this requires pk_i and, thus, is not anonymous.

To anonymize this, [325] utilizes a trapdoor permutation, TRP. Each P_i has a public value associated with it for each possible tag, $V_i = H(i|tag)$, as well as a public key $TRP.pk_i$ for a trapdoor permutation f. Party P_i checks whether they are a leader for tag by using their private trapdoor key $TRP.sk_i$ to calculate $v_i = f^{-1}_{TRP.sk_i}(V_i)$ and then checks if v_i is less than the relevant threshold. If so, they compute a zk-SNARK that proves they know a preimage of one of the V_i that makes them eligible.

To summarize this section, there are certain desirable properties that proof-of-stake leader election should have:

- The selection function should resist stake grinding. That is, stakeholders should have minimal flexibility when it comes to computing their eligibility query. This often necessitates having restrictive rules on timestamps.
- The stake distribution used for leader election should be fixed prior to generating a random seed to prevent the specific form of stake grinding that comes from an adversary adaptively choosing their public keys.
- Ideally, the selection function should be privately evaluated, such that only the stakeholder knows that they are elected before they announce this to the network. This can be done by making the selection function depend on the stakeholder's secret key, most commonly using a VRF. This is necessary for preventing adaptive corruptions.

- The selection function should be fair in that a stakeholder's chance of being elected leader is proportional to their share of the total system stake. Part of this involves minimizing the bias that an adversary can force into the random seed. This bias can be sufficiently bounded by allowing block producers to include random data in their blocks and having enough blocks in an epoch such that – with high probability – at least one block producer is honest.
- Selecting multiple leaders in a time slot can lead to forks, so it is better to have only a single leader per slot. VRFs can lead to multiple leaders, but SSLE protocols can solve this.
- Protecting the anonymity of the block producer would be ideal, but most existing leader election procedures do not provide this property.

12.2. Leader Predictability and Security

Every proof-of-stake scheme discussed thus far suffers, to some extent, from the problem of predictability. When proof of work is used, a miner does not know that they are a block producer until the moment they have succeeded in generating a proof. In contrast, a proof-of-stake system necessarily results in stakeholders being aware of their right to produce a block a minimum of one block in advance. This advance knowledge of leadership slots provides important information to an adversary, which can help them more confidently engage in selfish mining/staking, double-spending, and bribery attacks [327, 328]. Before explaining these attacks, some additional definitions are useful. In the following, a block denoted $Pred(B)$ is the predecessor of block B, and $Pred^D(B)$ is the D-th predecessor block of B.

- **D-locally predictable**: A unit of stake s has this property if, at block A, the owner of s knows that they will be able to produce a block B after D blocks. That is, after seeing A, the owner of s knows that they can produce a block B such that $Pred^D(B) = A$.
- **D-globally predictable**: A unit of stake s has this property if, at block A, every stakeholder knows that the owner of s will be able to produce a block B after D blocks. That is, after seeing A, all participants know that the owner of s can produce a block B such that $Pred^D(B) = A$.
- **D-recent**: This is a negation of local predictability. A unit of stake s has this property if, at block A, the owner of s does not know whether or not they will have the right to produce a block B such that $Pred^D(B) = A$. The validity of block B depends on some information contained in the most recent D predecessors of B.

All proof-of-stake protocols, including BFT-based ones, are (at least) 1-locally predictable. This is necessarily the case because all information used as input into the leader election procedure must be agreed upon prior to the leader election. Additionally, for any D and any block A, a given unit of stake is either D-locally predictable or D-recent. Protocols with recency have security risks in that competing chains may use different random seeds for leader election, which can make it more challenging to detect certain malicious protocol deviations. As a result, there are security trade-offs between protocols with recency compared to those with local predictability.

As mentioned above, predictability facilitates more advanced versions of selfish mining, double-spending, and bribery. In protocols with global predictability, selfish mining and double-spending attacks can be conducted in a way that is guaranteed to succeed. With local predictability, these attacks are not entirely risk-free, but the adversary is still provided with a significant statistical edge due to knowledge of the attacker's future blocks. Luckily, global predictability is not necessary. The use of VRFs allow leader election to be privately evaluated and publicly verified. That said, globally predictable protocols do exist, so the security concerns relating to them cannot simply be dismissed.

Predictable selfish mining and double-spending operate almost identically, so the following description of selfish staking applies to both (selfish mining was introduced in Section 9.4). Let block A be the current best chain tip on a protocol with D-global predictability and t be the current timestamp, and let $S()$ be a score function for a chain (e.g., $S(B)$ may be equal to the number of predecessor blocks in a longest chain protocol). Globally predictable selfish mining works as follows [327]:

1. For all $k \in \{1,...,D\}$, let t'_k be the earliest time that the adversary is entitled to produce a block B such that $S(B) > S(A)+k$ and where the adversary is the block producer of all blocks between A and B.
2. For all $k \in \{1,...,D\}$, let t^*_k be the earliest time that the rest of the network is able to produce a block B such that $S(B) > S(A)+k$ and where the adversary is the block producer for none of the blocks between A and B.
3. At time t, if a k exists such that $t'_k < t^*_k$, the adversary should immediately stop publishing blocks until t'_k, at which point they publish B and every block along the path from A to B from the first step. If multiple k exist where this holds, the strongest attack is to use the largest one.

This works exactly like selfish mining in the proof-of-work case, except that all of the risks are eliminated for the attacker. An adversary only withholds blocks from the network when they know for sure that the attack will succeed. Predictable double-spending works the same way, except for the details of including the specific transactions on each side of the fork. With local predictability, the attacker cannot perform the second step but can choose to perform the attack when they know they will produce a disproportionately large number of blocks in a certain window to maximize their chance of success. These prediction attacks were studied as applied to the Tezos blockchain, where it was found that it is often profitable to create two-block reorgs for selfish mining and that opportunities for predictable double-spending could be frequent. An adversary with 36% of the stake could expect to perform a predictable 20-block reorg once per year, and a 40% stakeholder could expect an opportunity for a 20-block predictable double-spend on a daily basis [329, 330].

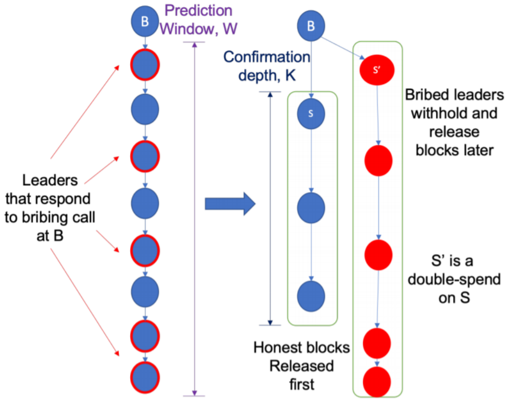
Using a VRF for leader election, as Ouroboros Praos does, provides security against adaptive adversaries. However, the adaptive adversary model used in this and other protocols fails to account for bribery attacks, such as the one depicted in Figure 38a [328]. Consider a proof-of-stake protocol with D-local predictability, where a merchant – and potential victim – considers a transaction final when the block it is included in has κ confirmations. First, consider the typical case where $D > \kappa$. An adversary can announce to the world that they would like to bribe block producers to contribute to an adversarial chain, and block producers can use their VRF proofs to demonstrate their eligibility. If the adversary can get $\kappa + 1$ eligibility proofs, they send a transaction to the victim merchant that is included in the next block. The attacker allows the honest chain to grow by κ blocks until the merchant provides whatever the adversary purchased. With goods in hand, the adversary creates a double-spend transaction and uses the $\kappa + 1$ eligibility proofs to publish a longer chain. For this attack to work, it is only necessary that $\kappa + 1$ out of the next 2κ block producers participate, but each of those block producers can have an infinitesimally small amount of stake. In addition, the bribed block producers have plausible deniability, as they need not sign conflicting blocks.

A naive solution to this problem would be for merchants to increase their required number of confirmations to something beyond the prediction window ($\kappa > D$). Note, however, that the security of these proof-of-stake protocols depends on generating random seeds with minimal bias. If the majority of block proposers in an epoch are bribed, the adversary can control the random seed to perform stake grinding and enlarge the prediction window into future epochs.

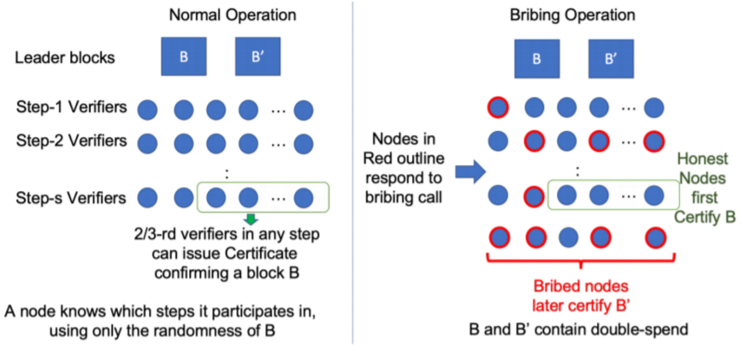
While BFT-based proof of stake may appear to be largely immune to this kind of attack, this is not necessarily true. Consider Algorand, where Byzantine agreement must occur over every block, each step of the agreement protocol has a separate committee, and the committees are 1-locally predictable. An adversary who can bribe $\frac{2}{3}$ of the committee at any given step can sign a block that conflicts with the one that honest stakeholders agreed on in an earlier step, as shown in Figure 38b. Because Algorand has relatively small, constant-sized committees, this $\frac{2}{3}$ supermajority may represent only a small fraction of the total stake.

Revisiting the leader election schemes discussed in Section 12.1.3, one can see that both Chains of Activity and Snow White have global predictability, while Ouroboros Praos and Algorand are locally predictable, though with very different predictability windows. Protocols with (short) recency face a different security obstacle: the undetectable nothing-at-stake attack, shown in Figure 39.

Let A be the highest scoring chain tip that an adversary is aware of in some D-recent protocol, and assume that the adversary has spread their coins across many public keys. To perform an undetectable nothing-at-stake attack, the adversary finds a block A', where $S(A')$ is maximal over all blocks that are not descendants of $Pred^D(A)$ (i.e., A' is the "next



(a) Chain-based



(b) BFT-based

Fig. 38. Predictable bribe attacks against proof of stake. [328]

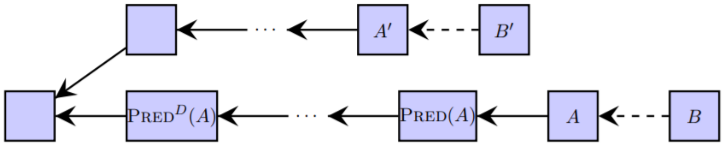


Fig. 39. Undetectable nothing-at-stake attack. The attacker creates both B and B'. [327]

best" block after A). Next, the attacker simultaneously tries to produce blocks B and B' on top of A and A', respectively. Should they successfully produce block B, they broadcast it. If they successfully produce B', they broadcast B' as well, so long as doing so would not cause a provable deviation. The two types of provable deviation occur when the same unit of stake is used to:

1. Sign two blocks, B and B', that have the same timestamp or
2. Sign two blocks, B and B', where the timestamp of B' is greater than that of B ($t_{B'} > t_B$), but $S(B) > S(Pred(B'))$. This is a provable deviation because if $S(B) > S(Pred(B'))$, an honest stakeholder would not produce B' at time t_B.

Because the protocol is D-recent, whether a valid B' can be produced on top of A' using a given unit of stake depends on blocks between A' and $Pred^D(A')$. As a result, each unit of stake provides another opportunity to produce a block on top of A'. Checkpointing protects against this.

The existence of this nothing at stake attack concretely lowers the security bounds of proof-of-stake protocols with recency. For example, a protocol called Nakamoto-PoS has recency and is secure against adversaries with less than $\frac{1}{1+e}$ ($\approx 26.9\%$) of the total stake [328]. In Nakamoto-PoS, the random seed is updated with every block and consists of the hash of the predecessor block with leader election using a VRF that takes the timestamp and previous block hash as input. Given that the randomness depends on the block and lacks consensus, the adversary can take advantage of the nothing-at-stake phenomenon to lower the security threshold. At the same time, this minimizes predictability, such that stakeholders only know if they can produce a block upon seeing the previous one. Note that, for predictable proof-of-stake systems like Snow White and Ouroboros Praos where the same random seed is reused for an extended period, asymptotic consistency guarantees match that of Bitcoin and Nakamoto Consensus, despite the nothing-at-stake problem [331, 332].

Interestingly, [328] showed how to adjust the Nakamoto-PoS protocol to create a family of chain-based proof-of-stake protocols that decouple the prediction window D from the security parameter, κ. This allows for dramatic reductions in the predictability window compared to algorithms like Ouroboros Praos and Snow White, where the epoch length (and thus prediction window) is proportional to κ. This family of protocols are defined by two parameters: c and s. The block hash used for randomness is updated every c blocks (so standard Nakamoto-PoS has $c = 1$), and only coins that are s blocks deep count toward the stake distribution used for leader election. Instead of following the longest chain rule, stakeholders use a fork-choice rule inspired by Ouroboros Genesis (Section 13.1.3). When comparing two chains, the stakeholder finds the block where the two chains diverge and prefers the chain where s blocks were produced after the fork most quickly based on their reported timestamps. If fewer than s blocks have been produced after the fork, then the longest chain rule is used. As c increases, the fraction of adversarial stake that the protocol can tolerate increases toward 50%, though predictability increases as well. This family of protocols can achieve any level of predictability by setting c to the desired predictability window while simultaneously being secure for any confirmation depth κ by using an appropriate value for s.

Recency is also problematic for proof-of-stake protocols that use the GHOST fork-choice rule (Section 11.2) because an adversary with little stake can dramatically increase the size of a particular subtree by trying to produce new blocks on top of every existing block in the subtree, magnifying the nothing-at-stake attack [327]. This subtree will grow exponentially faster than the number of blocks produced by honest stakeholders, forcing honest participants to accept the attacker's subtree.

12.3. Wealth Concentration, Block Rewards, and Centralization

In a proof-of-work cryptocurrency, miners face electricity costs and other operational expenditures, which often requires them to sell the cryptocurrency on the open market to fund their operations. This has the beneficial side-effect of distributing the coins to the public in a relatively "fair" way. However, when proof of stake is used, these costs no longer exist. As a result, there is a perception – real or imagined – that proof of stake faces the problem of wealth concentration, or "the rich getting richer." This section explores the consequences and likelihood of such a wealth concentration issue.

In a proof-of-stake network, the resource used to provide Sybil resistance is the native asset of the system. As a result, the block reward schedule (i.e., the monetary policy of the network) has potential implications for the security of the network beyond that of similar proof-of-work schemes. It is conceivable that under some reward schemes, large stakeholders are more likely to capture greater and greater shares of the total stake, whereas others do not encourage centralization in this way. Whether a particular reward scheme induces wealth concentration depends on the specific consensus algorithm. There may be differences when delegation to stake pools is allowed, if offline stakeholders exist, if a minimum bond must be posted in order to stake, and so on. Different studies investigate different models, and this remains a relatively understudied area, so care must be taken when trying to draw conclusions regarding the problem of wealth concentration.

In one study on this issue, Fanti et al. define the notion of *equitability*, which quantifies the degree to which a stakeholder's share of the total stake at some point in the future can change under a given reward function compared to their initial investment [333]. When a reward function is more equitable, the variability in this stake fraction should be small. The model used in [333] assumes that all stake is always online and available to produce a block when elected and that all rewards are reinvested into staking. When this is the case, they found that:

- The rich do get richer because larger stakeholders have lower variance in their investment returns than small ones, which allows their stake to compound. Allowing stake pools can lower this variance and, thus, the impact of compounding wealth.

- A large initial stake should be allocated in order for rewards to be more equitable. That is, the block reward should be small relative to the initial stake.
- The most equitable reward function is a geometric reward function where a constant fraction of the total stake is emitted at each block because rewards are smallest when the total system stake is smallest. Rewards then grow proportionally to the total system stake, thus bounding the wealth-compounding impact of each block. This implies an ever-increasing inflation rate. A constant reward function resulted in low equitability, and a decreasing reward function was even worse.
- Strategic behavior, such as selfish staking, exacerbates equitability issues more under proof of stake than under proof of work due to this compounding effect (even before considering predictability issues, as discussed in the previous section). Even the geometric reward function failed to mitigate the impact of compounding in the presence of selfish stakers.

Rosu and Saleh arrive at very different conclusions under a similar model but where the time horizon is infinite rather than finite [334]. They argue that while it is true that larger stakeholders are more likely to be elected and thus increase their share of the total stake, their share is decreased by an even larger amount when they are not elected. The result of these two competing forces is that a stakeholder's share is not expected to change over time from its initial value. While a geometric reward may minimize the variance of a stakeholder's share of the total stake over a finite period, it has high variance over large time periods and leads to wealth concentration in the limit. Contrary to [333], Rosu and Saleh predict that constant and decreasing reward functions induce a stable wealth concentration. Both analyses agree that a large initial stake total results in more stability for stakeholder shares.

Irresberger also investigated the evolution of share distributions resulting from different reward functions [335]. In this model, not all coins are actively staked, and participants must choose to either stake their coins or have them available for spending. Under these circumstances, the geometric reward schedule (called "constant" in [335]) results in greater centralization of stake than a dynamic reward schedule that targets a specific participation rate in consensus. That is, the least centralizing reward scheme is one that picks a certain number of stakers and increases the rewards when there are fewer or decreases rewards when there are surplus stakers. However, the difference between the geometric rewards and dynamic rewards was small as long as the geometric inflation rate remained low (1-5%). When block rewards decrease over time, the number of staking nodes tends to decrease as well and lead to significant wealth centralization.

The idea of having a dynamic reward schedule that adjusts in order to target a specific number of block producers has other advantages. In particular, because the capital used for staking is fairly liquid, the return on investment to staking must be competitive with alternative on-chain uses for that capital, such as collateralized lending protocols (i.e., decentralized finance [DeFi]) [336, 337]. If yields from these alternatives become higher than the yield from staking, "bank runs" can occur, where a significant number of stakers "unstake" their coins at around the same time, leading to a collapse in the security level of the system [336]. This is an especially significant concern in systems with decreasing block rewards or low inflation levels. A dynamically adjusting staking yield or a withdrawal delay for "unstaking" a participant's coins (as is done in Ethereum 2.0, discussed in Section 13.2) can help prevent this collapse. Interestingly, [337] found that both wealth concentration effects and the security risk from higher yield lending can be mitigated using *staking derivatives*. Consider a (fictional) asset, STK, that is used in a proof-of-stake system. A staking derivative is a synthetic asset, say, sSTK, that a staker can acquire while borrowing against their staked STK tokens. For the stakeholder to recover their STK and any block rewards they have earned from staking, they must buy back the STK with sSTK through an on-chain smart contract. Unfortunately, the positive results in terms of wealth distribution and security only hold when there is a non-negligible risk of having one's stake slashed, which is also undesirable.

Thus far, reward schemes have only been discussed for chain-based proof-of-stake schemes, where only a single stakeholder ultimately produces a block in a given time slot. For BFT-based proof-of-stake schemes, where an entire committee of stakeholders are involved in producing each block, incentive design becomes even trickier, as some of the committee members may be faulty and undeserving of reward. It is possible to design a fair reward scheme for this kind of system, where fairness means that any honest stakeholder should receive a fraction of the total reward equal to their fraction of the total stake, but only if the network is synchronous [338]. Without synchrony, it is not possible to determine whether a committee member is Byzantine or experiencing network delays, so rewarding stakeholders based on merit is not possible. Weighted voting – where the past behavior of faulty processes that do not send messages is used to assign them less influence in the consensus process – can further improve fairness but also induces centralization [339].

Another complication with some of this research on wealth concentration is the assumption that every stakeholder is actively staking and online at all times. There are several reasons why this is unlikely to be the case. First, without some form of stake delegation or stake pools, most individuals are not likely to keep an application up and running at all times due to sheer inconvenience. Second, being a block producer will usually require running a fully validating node, which may be performance intensive and contribute to the inconvenience of participation. Third, even with delegation or stake pools, it is likely that some (perhaps small) portion of users will fail to delegate or join a pool. In particular, it is likely to be small stakeholders who fail to overcome the friction involved because their expected absolute returns to staking are small. Finally, individuals who are less financially well off are likely to need or want greater liquidity from their assets, so these individuals may not be able to stake in systems where they must post a bond to become a block producer. Of course, only those who actually stake will get block rewards, so to the extent that some are excluded from staking, it is likely the case that wealth will tend to grow more concentrated

Network	Reward	Adj. Reward	Staked Value	Stake Ratio
Ethereum	4.79%	5.01%	\$35.8 B	15.62%
Cardano	3.23%	0.16%	\$9.5 B	68.04%
Solana	6.47%	–1.05%	\$8.3 B	72.83%
BNB Chain	2.66%	8.3%	\$7.5 B	15.35%
Avalanche	7.96%	2.32%	\$4.5 B	61.27%
Polygon	7.34%	3.89%	\$4.1 B	39.93%
Polkadot	14.34%	6.66%	\$4.0 B	47.03%
Tron	3.77%	1.69%	\$2.6 B	42.97%
Cosmos	23.1%	4.53%	\$2.6 B	66.97%
Internet Computer (Dfinity)	7.41%	–2.4%	\$1.8 B	73.2%
NEAR	9.24%	4.19%	\$1.0 B	45.34%
Tezos	5.32%	0.92%	\$774 M	72.41%

Table 1. Percentages of eligible tokens actively staked as of April 4, 2023. Adjusted reward is the annualized reward rate adjusted by the expected inflation of the network supply. [340]

within these systems. Table 1 shows the percentage of eligible tokens that are actively staked in some of the proof-of-stake networks with the greatest total value staked as of April 4, 2023.

This lack of universal staking suggests that stake pools may help reduce wealth concentration effects that could otherwise result from proof of stake (though care must be taken to avoid "address malleability" attacks if doing so [341]). Of course, this creates the need for reward schemes that properly incentivize the creation of a decentralized network of stake pools, which is a non-trivial problem. In fact, a "fair" scheme that rewards pools proportionally to their size/stake leads to an equilibrium with a single dominating stake pool [342]. Instead, [342] proposes a "cap-and-margin" reward scheme that has a Nash equilibrium resulting in the desired number of pools forming if participants are rational. If k stake pools are desired, the reward function caps the rewards for a pool once it reaches a size greater than $\frac{1}{k}$-th of the total stake in the system so that there is no longer an incentive for users to stake with a large pool. The reward function is also influenced by the amount of stake contributed by the leader of the pool, such that the pool leader will get higher returns by contributing more of their own stake to the pool. This is intended to discourage Sybil attacks, where a single entity with low stake is a leader for many different pools. There is a parameter that controls a trade-off between this Sybil-resistance property and the *egalitarianism* of the system, such that more Sybil-resistant rewards imply a less egalitarian reward scheme.

The egalitarianism of a system is a metric proposed in [343] based on the idea that a certain investment in capital to become a block producer should generate returns proportional to the capital invested. That is, in expectation, wealthier investors should not be able to gain disproportionate rewards compared to poorer investors. Egalitarianism is distinct from the equitability metric mentioned earlier in this section. Equitability captures the idea that over a series of blocks, a stakeholder's share of the total stake should not vary significantly from where it began. Egalitarianism, on the other hand, is about minimizing the variation in expected returns given an initial capital distribution. Stated differently, the randomness involved in equitability is over an execution of the protocol and how stake shares evolve, whereas the randomness involved in egalitarianism is over the distribution of wealth at the beginning of the protocol execution. Stake pools likely improve equitability by eliminating the penalty that comes from being offline but reduce the level of egalitarianism.

Proof of stake is more egalitarian than ASIC-resistant proof of work, which is more egalitarian than proof of work using ASICs [343]. This reflects the greater economies of scale that are possible under proof of work. In this respect, proof of stake has an advantage over proof of work in the likelihood of centralization, though egalitarianism does not quite capture the "rich getting richer" phenomenon the way equitability does. One reason for this is that the wealthy can afford to stake a much greater percentage of their assets than individuals with fewer resources (who need more liquidity). Since the egalitarianism metric is the return on capital invested and not the ability to invest capital in the first place, it does not account for this. Ultimately, the two Sybil-resistance mechanisms are simply different, and there is not a long enough history of deployment of either mechanism to determine whether one or the other is "better" at avoiding centralization. A few points of comparison are in order:

- In both proof of stake and proof of work, incentives matter for security and decentralization. However, the specific details of the incentives under proof of stake have greater significance and are less well understood. For instance, preventing the nothing-at-stake issue is easier when block rewards are small, but the research presented in this section is inconclusive regarding the inflation schedule.

- Proof of stake is far more dependent on the initial stake distribution for security than proof of work is. Achieving a good distribution is challenging both for technical reasons as well as a lack of understanding of what a "good" distribution would be in the first place. One way to overcome this is to have a proof-of-work phase where the cryptocurrency is first distributed "fairly" before switching to proof of stake. In practice, it is common for the cryptocurrency to be auctioned off when the system is initiated, but this is an inherently centralized process.

- It is usually assumed that the Sybil-resistance resource (work or stake) is acquired by honestly paying for it. In proof of stake, this should increase the exchange rate of the asset and the cost of the attack (this price increase may raise the cost of the attack by 20-40%, depending on several factors [344]). However, there are ways for well-positioned malicious parties to acquire the resource for free or cheap. In proof of work, the physical hardware used for mining can be stolen (or the owners of the hardware coerced). Depending on the situation, the attacker may still need to pay for operational costs, such as electricity. In proof of stake, the private keys used for staking can be hacked or stolen (including from vulnerable smart contracts), at which point there are no ongoing costs to use the stake. Identifying targets for such attacks in proof of work can involve monitoring electricity usage to detect large mining farms. In proof of stake, some large stakeholders can be easily identifiable: exchanges are likely to be large stakeholders and have the personal information of other large stakeholders. That said, it is likely easier to stake at significant scale without being detected than it is to mine undetected. On the other hand, the need for hardware in proof-of-work systems may add considerable friction for attackers that may result in the attacker being detected.

Apart from these differences, the consequences of a concentration of wealth or consensus power in these networks can differ. For example, both Nakamoto Consensus and chain-based proof of stake can recover from adversaries that gain a temporary majority of the Sybil-resistance resource, whereas BFT-based proof of stake is unable to recover [228]. In chain-based proof of stake, the spike in adversarial stake must be shorter than an epoch length in order to recover. Here, "recover" means that the ledger properties of persistence and liveness return after some period of insecurity. Of course, with proof of stake, any temporary majority can become a permanent majority if the majority behaves in any way that does not result in their stake being slashed.

Slashing is one of the biggest advantages of proof of stake compared to proof of work. Consider a situation where a powerful adversary manages to acquire a majority of the Sybil-resistance resource and has the capital to maintain this advantage for a while. For a proof-of-work system to survive, the proof-of-work algorithm must be changed in a hard fork. If the network used an ASIC-resistant algorithm when it forked, the attacker can use the same equipment to attack the hard-forked chain. If the algorithm was mined by ASICs, then a hard fork would resolve the attack by making all of the attacker's hardware useless, but this nuclear option destroys all of the honest miners' hardware as well. Proof of stake with slashing is better here because for the most serious kinds of misbehavior – such as a block producer equivocating and producing conflicting blocks at the same height – the protocol can specifically target the adversary's stake for slashing. This is like burning down an entire mining farm in proof of work but avoids collateral damage. Slashing requires extra care in situations where the network has a hard fork that causes a chain split because two blocks may then be signed at the same height on both sides of the fork and slashed.

13. Proof-of-Stake Protocols

13.1. Chain-Based Proof of Stake

13.1.1. Chains of Activity

An early chain-based proof-of-stake proposal is the Chains of Activity (CoA) algorithm that was designed for the UTXO model [311]. Unlike many of the proof-of-stake protocols described later in this document, the security of CoA is heuristic rather than proven. It borrows the *follow-the-satoshi* idea from the proof-of-activity hybrid algorithm described in Section 14.2 and introduced in Section 12.1.3. To "follow" a satoshi, one takes an index of the atomic unit of cryptocurrency, finds the block in which that unit was minted, and then traces the transaction graph from its minter to the public key of whoever currently owns that unit. Given some suitable procedure for securely generating that initial index, validators are able to agree on a leader. The CoA protocol utilizes the following parameters:

- There are 2^{κ} satoshis minted in the system.
- $w \geq 1$ is a subgroup length.
- $L = \kappa * w$ is a group length .
- $comb()$ is a function with L bit domain and κ bit range. It can be as simple as concatenating the inputs, but a variety of functions are possible.
- G_0 is the minimum block interval time.
- T_0 is the double-spending safety bound.
- C_0 is the minimal stake amount.
- C_1 is an award amount such that $0 \leq C_1 < C_0$.

The protocol itself can be described by assuming that some leader with a publicly known identity has been elected via some mechanism, and they construct and sign block B_i. Each block B_i is associated with a deterministically generated, uniformly distributed bit b_i (e.g., the least significant bit of a hash of B_i). For two blocks B_i and B_j, the time gap between them must be at least $|j-i-1| * G_0$, and a newly created block is considered invalid if the timestamp is too far in the future compared to the local clock's time. It is possible that a leader elected for a given slot is inactive, in which case their slot would be empty and have no block associated with it.

An epoch in CoA consists of L valid blocks $B_{i_1}, ..., B_{i_L}$ being created. After an epoch is completed, the network forms a κ-bit seed $S^{B_{i_L}} = comb(b_{i_1}, ..., b_{i_L})$. The seed is then used to derive the identities of the block producers two epochs later (i.e., skipping L blocks) using follow-the-satoshi. Specifically, if the next L blocks are $B_{i_L+1}, B_{i_L+2}, ..., B_{i_L+L}$, then the stakeholder elected to create block B_{i_L+L+z} is the one found by following the satoshi with index $H(i_L, z, S^{B_{i_L}})$ for $z \in \{1, ..., L\}$. If a node is ever presented with multiple chains, the chain with the most blocks is canonical.

CoA uses *slashing* to penalize bad behavior by forfeiting the stake of a malicious validator when misbehavior is detected. It may be the case that the satoshi used in leader election belongs to an unspent output with $c < C_0$ coins, or below the minimum. In this case, the leader must submit an additional signature proving ownership of the difference, $C_0 - c$ coins, before creating their block, and at least C_0 coins must be available to be slashed. The relevant outputs may not be spent for another T_0 blocks after the newly created one. The stakeholder who creates a block B_i may equivocate by creating another block at the same height, B_i', but any other leader elected in the next T_0 blocks can submit the conflicting signatures as a special transaction in their block to forfeit C_0 of the equivocator's coins, awarding the leader who caught them with C_1 of the forfeited coins.

There is also a "three strikes" rule to exclude inactive participants. If an output $txout_0$ was selected by follow-the-satoshi and has not created a block three times in a row, then $txout_0$ is no longer allowed to participate in the creation of new blocks (once the coins in $txout_0$ are spent, the new UTXO can be used for staking). Should follow-the-satoshi select $txout_0$, the slot is skipped and no block is produced. This helps CoA maintain liveness and performance even when users lose their coins or are offline for long periods.

The leader election process in CoA leaves a few things to be desired from a security perspective. First, as discussed in Section 12.2, having leader assignments known in advance is a security risk. An adversary who knows that they are the leader for several consecutive blocks may be able to perform predictable double-spends that succeed with certainty. Further, as discussed in Section 12.1.3, CoA results in leaders being *publicly* known in advance, so it cannot be secure against adaptive adversaries. This combination – but especially the predictability – can facilitate bribery attacks, though the slashing mechanism partially mitigates this risk. For a large epoch length L, this predictability problem becomes worse. On the other hand, a small L would make it easier for a malicious coalition to influence future leader elections. Finally, CoA requires a form of checkpointing to prevent long-range attacks.

13.1.2. Snow White

The first provably secure proof-of-stake system designs were Snow White [313, 314] and Ouroboros Praos (Section 13.1.3). Snow White is secure against *weakly adaptive* adversaries, where corruption takes place with some delay, whereas Praos is secure against fully adaptive adversaries. Snow White follows the longest chain rule and is secure if there is an honest majority of *online* stake but requires longer confirmation times than Nakamoto Consensus (simulations show 34% to 43% more blocks are required). From a networking perspective, Snow White follows the "sleepy consensus" model, which is similar to synchrony but allows nodes to go offline and start back up without losing security guarantees [28].

To address the issues of stake grinding and adaptive public key selection (to adversarially bias the random seed), Snow White uses a "two-lookback" mechanism, where each epoch's committee and random seed are generated in two separate phases with each phase lasting for roughly κ blocks, where κ is a security parameter. In the first phase, the prefix of the chain with the trailing 2κ blocks removed is used to determine the next consensus committee. In the second phase, randomness is extracted from the blocks in $chain[-2\kappa : -\kappa]$ to generate a random seed used for leader election in the current epoch. By forcing the committee to be selected κ blocks before the seed, an adversary cannot adaptively choose their public key based on the random seed to perform stake grinding.

When an honest validator creates a block, they include some random data in it. As long as a single honest block is produced in $chain[-2\kappa : -\kappa]$, randomness can be extracted from the chain in order to create the agreed upon random seed. This means that κ should be set such that the chain quality property can ensure at least one honest block over the κ block period. Snow White uses the FruitChains idea (Section 11.3) to improve chain quality. This setup still allows an adversary to bias the seed slightly by using strategically chosen data in its own blocks at the end of an epoch, but they cannot gain a significant long-term advantage as long as the same seed is reused for leader election a sufficient number of times in a row.

The Snow White protocol frames its lookback parameters by clock time rather than blocks. Here, 2ω is a lookback parameter for determining the next committee, and it must be sufficiently large that alert nodes will have a common prefix in their local chains by the start of an epoch. Similarly, ω is the lookback parameter for determining the next epoch's random seed. Let $extractpks()$ be a function that takes a chain and determines a committee from it as a set of public keys. Similarly, $extractseed()$ takes a chain, outputs the random seed used for leader election, and may be as simple as concatenating the random data included in each block of the chain prefix.

The e-th epoch takes place during the time interval $[start(e), end(e))$, and lasts for a duration of T_{epoch} time steps. An alert validator determines the committee for epoch e by finding the last block of its local chain with a timestamp no later than $start(e) - 2\omega$. Denote the index of this block in the chain as L_0. The committee is the output of $extractpks(chain[: L_0])$. Similarly, at any time $t \in [start(e), end(e))$, alert validators determine the random seed by finding the last block in its local chain whose timestamp is no greater than $start(e) - \omega$ and denote its index as L_1. The random seed is the output of $extractseed(chain[: L_1])$. At any time step t, a validator determines if it has been elected the block proposer by checking whether $H^{seed_e}(pk, timestamp) < D_p$, where D_p is a difficulty parameter, H is a random oracle, and pk is the public key of an elected committee member. In proof of stake, if a public key has m units of cryptocurrency associated with it, then the block proposer check can be $H^{seed_e}(pk||i, timestamp) < D_p$ for $i \in \{1, ..., m\}$. The difficulty parameter is set such that a committee member is elected in a given time step with probability p. If elected, the validator extends its longest chain by signing a new block. Block timestamps must be strictly increasing, and validators reject blocks with timestamps in the future. The security of this scheme depends on the lookback parameters being sufficiently far in the past that alert nodes will share common prefixes up to at least block L_1 and that the parameters are sufficiently spaced apart such that $seed_e$ cannot be predicted until sufficiently long after the committee is elected.

Snow White uses checkpoints to handle posterior corruption attacks. New nodes or those offline for a long time need to be able to determine the correct history to believe in. They can do this by contacting a list of nodes (perhaps provided to them upon startup) – the majority of which are *alert* (i.e., honest and online) – and trusting the majority view in terms of stake. Further, alert validators will always reject any chain sent to them that would reorg "too many" blocks. Specifically, any chain that would revert $\kappa_0 = \frac{\kappa}{2}$ blocks is rejected by alert validators.

Given a sufficient posterior corruption window, even a majority of corrupt stake *before* that window should not be able to harm consensus, which implies that security requires that cryptocurrency not change ownership "too quickly" (the stake shift issue described in Section 12.1.1). The posterior corruption window, $W > \omega$, also relates to security against weakly adaptive adversaries. When an adversary knows the next seed used for leader election, they must not be able to adaptively corrupt leaders for at least one epoch length. Of course, such adaptive corruptions are possible in the real world. Security requires that for any committee, the alert stake must outnumber the adversarial stake for a time window of W. The protocol ensures that if a validator is corrupted after time $t+W$, they can no longer influence any of the chain's history before time t.

13.1.3. Ouroboros Family: Praos and Genesis

Ouroboros Praos was the first proof-of-stake protocol to be proven secure against fully adaptive adversaries [319]. Praos uses a VRF for adaptively secure leader election, which maintains unpredictability even against adversarially chosen keys. To address posterior corruption issues, Praos uses forward secure key-evolving signatures, where old keys must be securely erased after a short period (validators are trusted to perform this erasure, even though it may not be rational). This prevents an adversary from generating signatures for messages that were issued in the past. As with Nakamoto Consensus, Praos follows the longest chain rule and was proven secure in the bounded delay model assuming an honest majority of stake. Unlike with Nakamoto Consensus, Praos validators will not reorg more than k blocks of the chain, where k is a security parameter.

Praos divides time into discrete *slots*, only some of which have blocks created in them. Empty slots allow honest validators to remain synchronized. This is analogous to how Nakamoto Consensus requires the expected block time to be significantly longer than the time it takes to propagate the block across the network. The canonical longest chain will have at most one block per slot, though there may be multiple leaders elected during a particular slot. The use of a VRF ensures that only slot leaders themselves are aware that they are a leader in a given slot, and other validators are not until they receive signed blocks that include a valid VRF proof from the legitimate leader. The leader election process in Ouroboros uses the following parameters:

- R is the number of slots in each epoch.
- f is the *active slot coefficient*, or the chance that anyone is elected leader of a particular slot.
- ϕ_f is the function that – given a stake fraction α_i for user i – outputs the probability of being elected leader: $\phi_f(\alpha_i) = 1-(1-f)^{\alpha_i}$.

- k is the maximum reorg depth.
- η_j is the random seed for epoch j.
- s is the number of slots to check for chain density in Ouroboros Genesis.

The protocol is defined over a sequence of $L = ER$ slots, $S = \{sl_1, ..., sl_L\}$, consisting of E epochs with R slots in each epoch. For each new epoch e_j, the stakeholder distribution used to select leaders is drawn from the most recent block with timestamp up to $(j-2)R$ in the local canonical chain. Using the stake distribution as of two epochs in the past helps ensure agreement and, as with Snow White, prevents an adversary from adaptively choosing their public keys to aid in stake grinding. To generate the random seed for epoch e_j, a validator takes every block from the chain belonging to the first two-thirds of epoch e_{j-1}, concatenates the block randomness VRF outputs and proofs into a value v, and computes the new epoch randomness η_j as $H(\eta_{j-1}||j||v)$. Stated differently, validators use the first $\frac{2R}{3}$ blocks of the prior epoch to determine the current epoch's randomness using the entropy contained in each block from the leader election process itself.

With the stake distribution and epoch randomness determined, slot leader assignment works as follows. A stakeholder U_i is independently selected as leader for slot sl_j with probability p_i depending only on their relative stake, α_i. The active slot coefficient, f, determines the relationship between the probability and the relative stake. Specifically, $p_i = \phi_f(\alpha_i) = 1-(1-f)^{\alpha_i}$. Since $\phi_f(1) = f$, f is the probability that someone with all of the stake is the slot leader. This function is unaffected by whether a party splits up their stake into multiple virtual identities or not. The threshold for a stakeholder U_i for epoch e_j with a stake of α_i^j is $T_i^j = 2^{L_{VRF}}\phi_f(\alpha_i^j)$, where L_{VRF} is the length of the VRF output. Validators check whether they are slot leader by evaluating the VRF with the secret key associated with their stake, taking the slot number and epoch randomness η as input. They are elected to be a leader if the VRF output is beneath the threshold. The procedure described here is very similar to Snow White but protects against fully adaptive corruptions. It still allows some mild stake grinding via the hashing of VRF outputs, which an adversary may influence, but the bias that this allows is bounded with suitable parameter choices.

With leader election out of the way, the rest of consensus is simple: validators follow and build off of the longest valid chain they are aware of but refuse to overwrite more than k blocks of the chain. When a node is offline for a long time or is bootstrapped for the first time, it must acquire the chain or a checkpoint from a trusted party. The Ouroboros Genesis protocol was proposed in an effort to remove this trusted component and allow a new node to bootstrap themselves securely from the genesis block without needing to acquire a trusted checkpoint [345].

Ouroboros Genesis is nearly identical to Praos but modifies the chain selection rule. Let the local best chain be C_{max}. For each potential new chain C_i, if C_i forks from C_{max} at most k blocks, and $len(C_i) > len(C_{max})$, then set $C_{max} = C_i$. If, however, C_i forks from C_{max} by more than k blocks, validators apply a different rule. Let j be the highest slot index where C_i and C_{max} have a block in common and s be a parameter for the rule. Then, if $len(C_i[0:j+s]) > len(C_{max}[0:j+s])$, set $C_{max} = C_i$. That is, for some s-slot period after the point where the two chains diverge, validators compare the density of the two chains and prefer the one with more blocks shortly after the fork. During this short period after the fork, the two chains still share a view of how leader election should occur in the post-fork slots. Intuitively, an honest majority at that time will work on the chain that ends up becoming canonical, so the canonical chain should be the one that contains more blocks during that period.

This new chain selection rule is not without risk and may require re-parameterizing the system in ways that may be detrimental to security. This is because this new chain selection rule allows the following attack:

1. An adversary acquires a substantial amount of stake but still less than half. This adversary runs a modified version of the software that does not erase old private keys after evolving them.
2. At the start of each epoch, the adversary checks which slots they will be eligible for. If the number of slots they are selected for suggests that there is a significant chance of producing more blocks than everyone else combined during some s-slot period of the epoch, the adversary will observe the production of blocks but avoid creating any. When the s-slot period is over, the adversary will have near certainty that they could have overwritten the chain during that period. Let the first slot of this s-slot period be denoted sl_j.
3. The adversary may then wait any arbitrary period of time after the requirements from the prior step have been met. At any time, they may create an alternative chain forking from sl_j. This alternative chain will include every block that the adversary is entitled to (they can do this because they did not delete their keys). Because the chain density in the s slots after sl_j is higher in this attack chain, the network will reorg around the attack chain (even, say, 10 years after the fork occurred).

This attack will be possible if there is any adversary who can acquire more blocks than everyone else for any s-slot period in the lifetime of the system. It would permanently eliminate the security of the system in a way that may be undetectable to honest parties until after the fact. Therefore, the security of the system relies on having a parameterization that would make it vanishingly unlikely for a party possessing a "realistic" amount of stake to ever be in a position where they can reorg the rest of the network during any s-slot period. In particular, this requires the product $s * f$ – that is, the number of blocks that the system expects to be produced over s slots – to be large. Increasing f is equivalent to lowering the expected block time, which reduces the security margin due to latency issues (see Section 10.2.1). Alternatively, increasing s necessitates increasing the epoch length R. Longer epochs mean that chain-based predictability attacks become more severe (see Section 12.2), and there is a longer period for stake shift to reduce the security margin.

Ignoring this extra attack, Ouroboros Genesis comes extremely close to providing the same security guarantees as Bitcoin. However, it is still subject to posterior corruption if the erasure model does not hold, requires a secure initial distribution of stake, and relies on more advanced cryptographic primitives.

There are several other protocols in the Ouroboros family, but the details of their operation are out of scope for this document. Ouroboros Chronos extends Genesis in order to remove the need for globally synchronized clocks and, thus, remove dependence on the Network Time Protocol (NTP) that proof-of-stake protocols typically require [346]. Ouroboros Crypsinous is a design that ensures that provably secure proof-of-stake leader election can occur in a system that includes privacy-preserving transactions such that it is not clear from the ledger how many coins are associated with each public key [323].

13.1.4. DFINITY

The DFINITY consensus protocol is composed of four layers: identity registration, a randomness beacon, the blockchain, and the notarization layer [347]. By combining these layers, DFINITY is able to offer some of the benefits of both chain-based and BFT-based proof-of-stake systems.

Identity registration requires potential validators to make a deposit of stake associated with their public key so that a known and agreed upon number of parties exists in order to perform the necessary threshold cryptography utilized by the protocol and so that misbehavior can be punished by slashing these deposits. These registered keys are then used to set up the randomness beacon by creating a VRF out of the BLS threshold signature scheme. A group of validators must run a distributed key generation (DKG) protocol to create a group public key for verifying the random outputs. To compute the random seed for round r, $seed_r$, members of the group provide partial signatures over the round number and $seed_{r-1}$. The partial signatures are aggregated into the final signature, which is then hashed to produce $seed_r$.

Since many users are expected to participate, the scalability of the system requires that only subsets of the registered users are involved in generating random beacon values (as well as participating in notarization, which is described later). A random seed is used as part of a cryptographic shuffle to select a list of m groups or committees. The committees run the DKG protocol, and the resulting public keys can be put in the genesis block. The round r committee used for the beacon and notarization, G^r, is the j-th listed group, where $j = seed_r \ mod \ m$. The members of group G^r create the random seed used to select group G^{r+1}.

In practice, it is unlikely that all identities will be registered before the system starts up, and allowing dynamic participation is desirable. To do so, special registration and deregistration transactions can be used to join or exit the system. Time is divided into epochs, where the first block of the epoch contains a summary of all the registrations or deregistrations from the prior epoch. In this new block, a random seed is used for a cryptographic shuffle as with the static case, and new groups can perform DKG and issue another special transaction to register their aggregated group public key. These newly registered groups can participate and be selected two epochs after they have been registered.

The blockchain itself is constructed using the *probabilistic slot protocol* and *notarization*. These layers assume that a group G^r and random beacon value $seed_r$ has been selected for round r. Any user $U \in G^r$ may produce a block in round r, but the probabilistic slot protocol establishes a *priority* among group members and favors high priority blocks. A cryptographic shuffle is performed using $seed_r$ to establish a ranked order among group members, where a lower rank means a higher priority. There is also a *weight* function $w(x) = 2^{-x}$, and a block B's weight is $w(B's\ rank)$. The weight of a chain is the sum of the weight of its blocks. Similar to other longest chain protocols, the heaviest chain wins in DFINITY.

As validators produce blocks in a round, they are sent to the group in charge of notarization for that round, which may be the same group. The notary waits for a protocol-specified $BlockTime$ to receive proposed blocks. After $BlockTime$, notary group members will sign all of the highest priority valid blocks that they saw in that time and continue to sign the highest priority blocks until the next round begins. The next round begins after observing a notarization for the current round, where a notarization is an aggregated threshold signature on the block. The notarized block is broadcast to the network, and validators will update their local chains, end the round, and have the random beacon output a seed for the next round. Because the notary is optimistic instead of a full consensus algorithm, it can notarize two conflicting blocks at the same height. This is resolved through the chain selection rule's weight function. The role of notarization is to make it impossible for an adversary to build a secret chain of notarized blocks, thus addressing nothing-at-stake issues and selfish mining. It also establishes finality for blocks, such that a finalized block can never be undone.

The finalization procedure involves gathering notarized block proposals and placing them into buckets based on their round. Once a notarized block is found in round r, finalization is scheduled for the blocks of round $r-1$ in T time units. At that time, the heaviest common prefix ending at round $r-1$ is finalized. If at round r, all valid, notarized blocks B_{r-1} point to the same predecessor, B_{r-2}, validators finalize B_{r-2} and its predecessors. Each observer can specify their own T; it need not be common.

The security proof for DFINITY assumes a synchronous network and is secure for an honest majority of stake against mildly adaptive adversaries. If Δ is the maximum network delay, then secure parameterization of DFINITY requires that $BlockTime \geq 3\Delta$ and $T \geq 2\Delta$. The DFINITY protocol was further analyzed in [348], which pointed out that if the highest priority member of a group is Byzantine, the communication complexity of the protocol is unbounded because they can send an unlimited number of highest-priority blocks, and honest validators are required to vote on all of them. To fix this, whenever an honest validator sees two blocks from the same block proposer, the conflicting signatures are used as proof of misbehavior and lower the effective priority of the block. This fix increases DFINITY's worst-case latency to 17Δ. In the optimistic case that the actual network delay is small compared to Δ, the expected latency is 8Δ. When the actual delay is Δ, the expected latency is 14Δ without equivocation.

13.2. Ethereum 2.0

The Ethereum network has recently undergone a transition from a pure proof-of-work system using GHOST as the fork-choice rule (Section 11.2) to a hybrid system that combines proof-of-work and a proof-of-stake finality layer (Casper FFG, described in Section 14.3.2) and, finally, to a pure proof-of-stake system dubbed Ethereum 2.0 [349]. This section may not completely match the "final" design for Ethereum's proof-of-stake system since it is under continual development and subject to change.

The architecture of Ethereum 2.0 includes a *beacon chain* that handles random number generation and coordinates separate *shard* chains. Sharding is a scalability technique in which each shard is its own separate blockchain with its own state (described in further detail in Section 15). It allows nodes to only validate a fraction of the system's total transaction throughput. The beacon chain stores validator information and establishes consensus over data related to the state of each shard while mediating cross-shard communications. To become a beacon chain validator, users must deposit 32 ETH into a staking contract so that their stake may be slashed in case of misbehavior. This amount was chosen to balance competing factors: larger deposits prevent people from participating, whereas smaller deposits increase the overhead of verifying the chain. The beacon chain divides time into 12 second slots, and each epoch has 32 slots (6.4 minutes). An "eek" is 2,048 epochs ($\approx$ 9 days). In each epoch, validators make *attestations*, which consist of:

- A hash of what the validator considers the chain tip
- A hash of what the validator believes is the correct shard block to include
- The "source" and "target" hashes from Casper FFG (see Section 14.3.2 for details)
- A signature from the validator over the above data

These attestations are used to come to consensus using a combination of a variant of the GHOST fork-choice rule called Latest Message Driven (LMD) GHOST and Casper FFG [350]. LMD GHOST is a proof-of-stake variant of GHOST where, at each fork, nodes prefer the fork that contains more total support from validators (based on the stake-weighted sum of attestations) while counting only the most recent message from each validator. The combined protocol (dubbed "Gasper") finalizes blocks, keeps track of the latest justified checkpoint using Casper FFG, and uses the LMD GHOST rule to determine the chain tip by treating the latest justified checkpoint as the root of the chain. The LMD GHOST rule is shown in Figure 40.

Validators are rewarded or punished based on their attestations. Correct attestations are

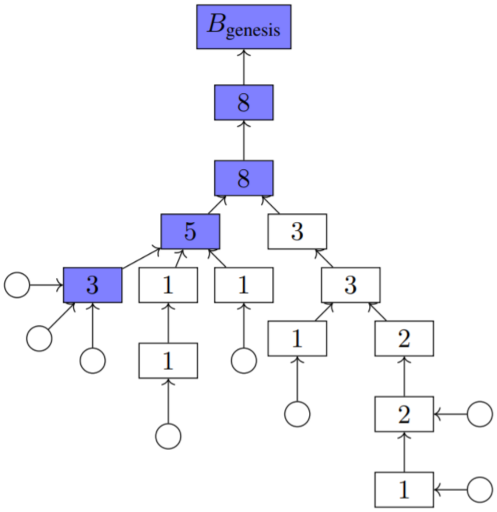


Fig. 40. LMD GHOST. The number in each block is the weight of a block when each vote has weight one (circles represent votes). The blue chain, ending with the block of weight three, is the canonical chain. [350]

rewarded, whereas missed slots result in penalties. The interest rate for correct voting depends on how many validators are participating. Fewer staking validators will raise the interest rate to entice more validators to secure the chain. If penalties accrue to the point where a validator has less than 16 ETH at stake, they are ejected from the validator set. Validators who equivocate by voting for conflicting blocks are slashed, lose some fraction of their deposit (substantially more than the penalty for missing a slot), are removed from the set of validators, and have the remainder of their deposit frozen for an extra four eeks before being allowed to withdraw their funds. During this waiting period after being slashed, the validator is penalized proportionally to how many other validators are slashed during that same period. This is to discourage correlated failures. Isolated, honest mistakes will not be penalized as heavily as active attacks using a large portion of the total stake. It also means that smaller validators take on less risk than larger ones and discourages validators from joining large stake pools.

Validators may also voluntarily leave the system and are allowed to withdraw their funds after 256 epochs, or about one day. If a large number of validators try to exit at once, a queue will form and they will exit over time. This queuing prevents an adversary from creating many validators, performing some malicious action, and then exiting before they can be slashed. Validator registration is similarly rate-limited.

Leader election in Ethereum 2.0 requires generating unpredictable random seeds using a commit-reveal mechanism inspired by a process called RANDAO. Ultimately, the random seed is used both to determine block proposers and to assign validators to committees. There is a beacon chain committee per slot, where one validator is the block proposer and the rest of the committee members issue attestations for the slot. There are also committees of validators responsible for handling each particular shard chain. A cryptographic shuffle is performed on the validator list using the seed as input, and committees are consecutive slices of the resulting list.

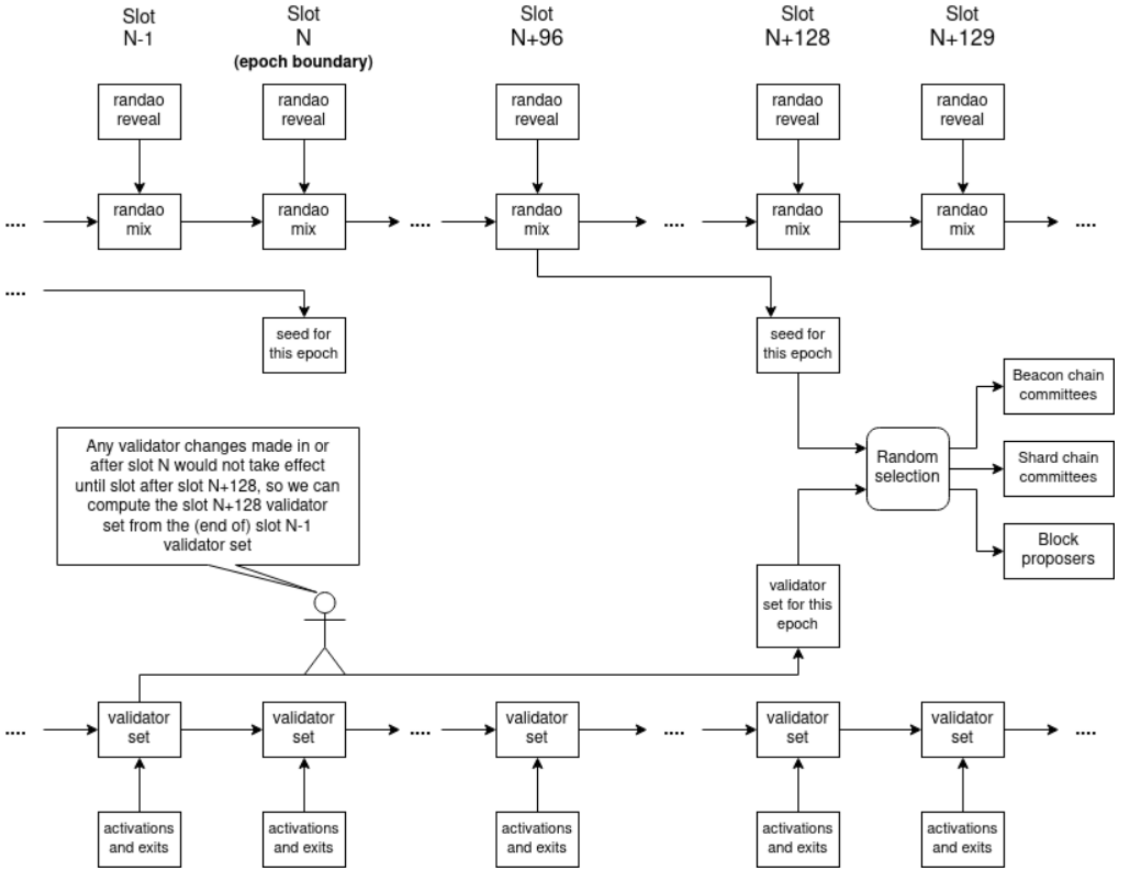
To prevent stake grinding via adversarial key selection, the validator set is fixed four epochs in advance. There is only a single block proposer per slot, and this proposer includes a *randao mix* into their block. The randao mix is a hash that is XORed into the seed to update it every slot. Specifically, it is the hash of a BLS signature over the current epoch. This hash is unknown to other validators ahead of time, but there is only one valid submission due to the uniqueness property of BLS signatures. The randao mix from the beginning of epoch e is used to calculate the seed in epoch $e+1$. Note that only the last proposer in an epoch has the ability to bias the seed by one bit, which they can do based on their decision to publish a block or not, potentially sacrificing their reward by withholding a block. This process is shown in Figures 41a and 41b.

To prevent even this single bit of bias that an adversary can induce, the output of the randao mix can be input into a verifiable delay function (VDF). The VDF's delay is parameterized to be longer than the time window where a validator could benefit from influencing the random seed, or at least one epoch. This prevents the final block proposer from being able to know the eventual seed quickly enough to decide whether to withhold their own randao mix or not. The Ethereum 2.0 randomness generation process favors liveness over being unbiasable because of the possibility of using a VDF as well as the economic penalty that comes from sacrificing the block it would take to manipulate the seed. As a result, the beacon chain can continue producing pseudorandom numbers even when many validators are partitioned from the network or offline. Contrast this with, say, DFINITY, where a group may fail to reach the threshold needed to produce the aggregate BLS signature, thus stalling the chain until the network is healed.

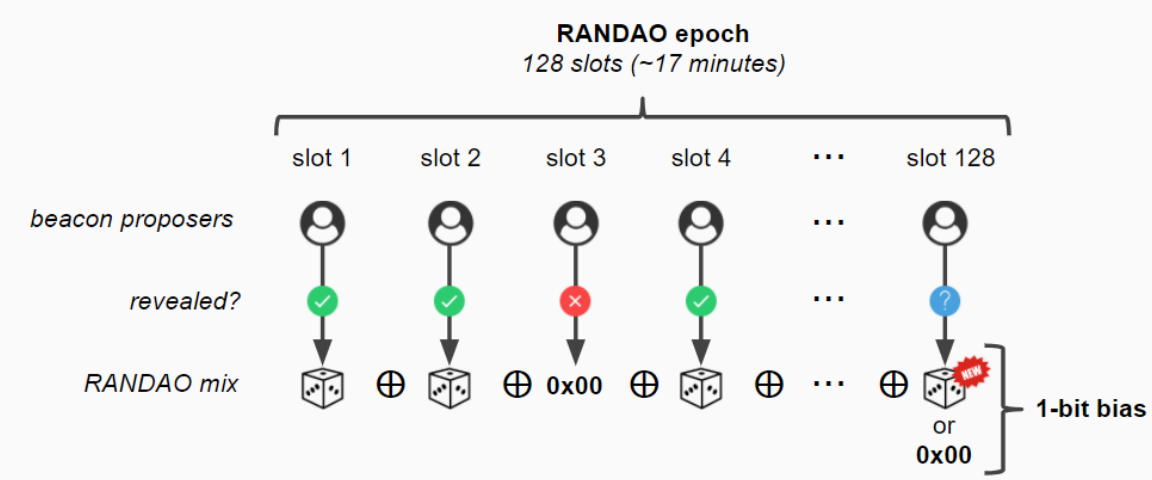
13.3. DAG-based Proof of Stake

13.3.1. Fantômette

Fantômette is a DAG-based protocol that was designed to be game-theoretically secure under the BAR model (see Section 1.4) so long as the non-altruistic stake is less than $\frac{1}{3}$ of the total stake [353]. Specifically, a coalition of less than $\frac{1}{3}$ of participants are unable to significantly increase their rewards if the rest of the participants are altruistic, and a coalition of up to one-fourth of the participants being Byzantine are unable to lower the payoff for altruistic participants. It employs several cryptographic tools, such as publicly verifiable secret sharing (PVSS), a verifiable random function (VRF), and a verifiable delay function (VDF). In Fantômette, the DAG is formed using two kinds of references: each block has a single parent block, denoted B_{prev}, but can also reference a set of other chain tips, which they call *leaves* and denote B_{leaf}. Fantômette is inspired by the Phantom protocol, and its description utilizes much of the same language employed in Section 11.5.2. Some additional definitions are used as well:



(a) Ethereum 2.0 randomness



(b) RANDAO in Ethereum 2.0

Fig. 41. Ethereum 2.0 Randao architecture. Note that 128 slots would take 25.6 minutes rather than 17, based on the proposed slot duration of 12 seconds. [351, 352]

- $Ancestors(B)$ is the set of all blocks that are direct or indirect parents of B.
- $DirectFuture(B)$ is the set of blocks that directly reference B (that is, blocks where $B \in B'_{leaf}$ for any B'). Contrast this with $future(B)$, which includes blocks that indirectly reference B as well.
- The *biggest common prefix DAG* (BCPD) is the largest subDAG that more than half of the players agree on.
- *Double* is the set of all blocks that use the same proof-of-stake leader election eligibility proof but different contents. That is, these are equivocating blocks.

Fantômette uses a leader election protocol called Caucus that is designed to be secure against fully adaptive adversaries and is similar to Algorand's cryptographic sortition (Section 13.4.2). To be considered during the leader election process, participants are required to post a security deposit that may be slashed if misbehavior is detected. Caucus has four steps:

1. **Commit.** Participants must commit to their VRF secret key, sk, by issuing a special *commit transaction*, tx_{com}, that specifies their corresponding public key pk. These commitments are appended to a list of commitments, c, that is part of the system's public state, $state_{pub}$. After being added to c, a participant must wait for some fixed number of protocol rounds, x_{wait}, before they may be elected as leader. The added waiting time maintains unpredictability and prevents the adversary from grinding through adversarially chosen keys. Denote the round that a participant issued tx_{com} as rnd_{joined}.
2. **Update.** This step is run only on the first round, where $rnd = 1$. A certain threshold of participants must have successfully committed to the leader election. At this point, the participants run a coin-tossing protocol (using a PVSS) to generate a random value, R_1, and then update the system state to $state_{pub} = (c, R_1)$.
3. **Reveal.** Participants check their own eligibility in each round where $rnd > 1$. Let $y_{rnd} = VRF_{sk}(R_{rnd})$, n_{rnd} be the number of eligible participants that have waited enough rounds since their commitment round, and $target = \frac{H_{max}}{n_{rnd}}$. A participant is eligible if $H(y_{rnd}) < target$. If a participant discovers that they are eligible, they issue a special *reveal transaction*, tx_{rev}, that includes y_{rnd} and $p_{rnd} = p_{sk}(R_{rnd})$, where p_{rnd} is the VRF proof.
4. **Verify.** When a participant sees a transaction tx_{rev} from participant i, they check whether $VerifyVRF(R_{rnd}, y_{rnd}, p_{rnd}) = 1$ and that $rnd_{joined} > rnd - x_{wait}$. If so, then the public randomness is updated to $R_{rnd+1} \leftarrow R_{rnd} \oplus y_{rnd}$, and the verification is considered a success. Otherwise, the public state remains unchanged.

   In some rounds, there will not be any elected leader, or an adversary is elected but does not issue a tx_{rev}. If no participant reveals tx_{rev} in a given round, the public randomness is updated as $R_{rnd} \leftarrow VDF(R_{rnd})$. If an honest participant finds that they are not eligible in the reveal phase, they immediately begin computing the VDF. If they have not seen a valid tx_{rev} by the time the VDF is computed, they re-check their eligibility using the new VDF output as R_{rnd}. This maintains liveness.

There may also be more than one valid tx_{rev}. In this case, the winning leader can be the participant whose y_{rnd} is the lowest.

The Caucus protocol may be modified in order to improve fairness by ensuring that the same participant is not elected leader a disproportionate number of times. When computing the *target*, the system may use $\frac{n_{rnd}+1}{2}$ instead of n_{rnd}. In this case, the verify step would add a rule that the participant was not leader in the prior $\frac{n_{rnd}-1}{2}$ rounds.

The basic idea behind Fantômette is that participants use their stake to bet on the block that they believe has the highest *score*, where a block is considered a bet on its ancestor set, while simultaneously demonstrating that they are well-connected and honest by including other leaf block references.

Fantômette employs a distributed checkpointing scheme in order to finalize blocks. Beginning from the genesis block, a candidate for finalization is a block that bets on the genesis block or specifies the genesis block as its parent. Candidate blocks have a *rank*, where this first set of blocks (with the genesis block as parent) belongs to rank $rk = 1$. In a system with n participants, a block B is a *witness* for candidate block C if $\frac{2n}{3}$ participants have bet on C when considering the set $past(B) \cup \{B\}$. In this case, C is said to be *justified*. If, in turn, a block B_2 is a witness of B, it is said to be a *second witness* of C. When a block B bets on a second witness with rank rk, then B is considered a candidate for rank $rk+1$. As described, this procedure requires a fixed set of participants and would require adjustment to allow participants to join or leave during the checkpointing process.

Fantômette's fork-choice rule calculates the score of a leaf block B as the sum of the number of references made by each block in $past(B)$, except for equivocating blocks (those in the set $Double$). To decide which block to bet on, a validator computes the score of each block and sets the parent reference as the highest scoring leaf block. More formally, given a DAG G:

1. $w \leftarrow \emptyset$
2. For $B \in Leaves(G)$ do: for $B' \in past(B) \setminus Double$ do: $w[B'] = |B'[B_{leaf}]|$
3. $CW \leftarrow argmax_{B \in leaf(G)} w(B)$
4. In case of a tie, choose the block with the smallest hash: $B \leftarrow argmin_{B \in CW} H(B)$, and return B.

After computing the winning block B based on the fork-choice rule above, the validator checks their most recent second witness block and, to avoid long-range attacks, ensures that there is a candidate block associated with it in $Ancestors(B)$. They then check whether they are eligible using the VRF and, if so, produce a block with B as the parent and every other leaf they see as a reference. The block B is considered valid if:

1. $B_{prev} = ForkChoiceRule(past(B))$. That is, B bets on the block that the fork choice rule requires based on B's view of the DAG.
2. The creator of block B successfully proves their eligibility.
3. If a witness block exists in $past(B)$, then there is also a witness block in $Ancestors(B)$. In other words, a participant may not bet on a non-justified block if they are aware of a justified one.
4. If B bets on a second witness block B_s, then B also bets on a block in $past(B_s)$. That is, $Ancestors(B) \cap past(B_s) \neq \emptyset$.

Incentivization in Fantômette uses a function inspired by Phantom that labels each block as a winner, a loser, or neutral. If the fork-choice rule selects a block B, then B and blocks in $Ancestors(B)$ are labeled as winners. A block that bets on a winner is neutral, and winning and neutral blocks in DAG G form $BlueSet_G$. Similar to Phantom, for a parameter k, a block B where $|anticone(B) \cap BlueSet_G| \leq k$ is labeled neutral. If $|anticone(B) \cap BlueSet_G| > k$, then the block is a loser. Specifically, the labeling function with input a DAG G and output a labeling of the DAG M is:

1. Set:
   - $B \leftarrow ForkChoiceRule(G)$
   - $BlueSet_G \leftarrow BlueSet_G \cup \{B\}$
   - $M(B) = winner$
2. For $B_i \in Ancestors(B)$ do:
   - $BlueSet_G \leftarrow BlueSet_G \cup \{B_i\}$
   - $M(B_i) = winner$
   - For $B_j \in DirectFuture(B_i) \setminus Ancestors(B)$ do:
     - $BlueSet_G \leftarrow BlueSet_G \cup \{B_j\}$
     - $M(B_j) = neutral$
3. For $B_i \in G \setminus BlueSet_G$ do:
   - If $|anticone(B_i) \cap BlueSet_G| \leq k$, then:
     - $BlueSet_G \leftarrow BlueSet_G \cup \{B_i\}$

- $M(B_j) = neutral$
- else: $M(B_i) = loser$

4. Return M

Rewards are calculated over a period of some number of blocks, at which point the above labeling function is applied to the BCPD. To encourage participants to reference each others' blocks and have the DAG be well-connected, the reward that a block provides is proportional to the number of leaves it references: $rwd(B) = |B_{leaf}| * c$, for some protocol-defined constant c. There are also two constants for punishing misbehavior or poor connectivity: blocks labeled as losers will penalize their creator by pun, and malicious behaviors (like equivocation or not referencing one's own blocks) are punished by $bigpun$.

Fantômette's reward scheme is incentive-compatible, so participants have the highest utility by behaving honestly. Recall that the validity of a block depends on it being the one chosen via the fork-choice rule based on its past set. In addition, rewards are proportional to the number of leaves referenced in the block, and the score of a block also corresponds to the number of leaves it references. Because the participant must follow the fork-choice rule, the only way they can pick another block is by reducing the number of leaves it references. However, this makes the block more likely to be labeled as a loser (as it will have a larger anticone) and decreases its reward if it is not. Participants can also withhold their blocks, but doing so means they will not be referenced as frequently and are less likely to be labeled as winners.

13.3.2. Avalanche

Avalanche is a leaderless protocol that provides a partial ordering of transactions via a transaction DAG [354]. It is typically associated with proof of stake but is agnostic to the Sybil-resistance mechanism (there is even a proposal for using IP prefixes for this purpose [355]). Avalanche – or rather, the Snowball protocol that it is based off of – operates rather differently from other algorithms described in this document. Nodes arrive at decisions based on repeatedly sampling other nodes for their opinion and then being steered toward a common view as rounds progress. This structure provides considerable efficiency advantages: there is only $O(1)$ communication cost per round over an expected $O(log\ n)$ rounds for a given node.

In fact, [354] presents a family of "Snow" protocols that follow a similar structure. Of particular importance is the Snowball protocol, because Avalanche is actually composed of multiple Snowball instances organized over a transaction DAG, which reduces costs from logarithmic in the number of nodes to constant and results in very low latencies for transaction settlement. As with several other DAG-based protocols, the algorithm is framed as deciding between two different colors: red and blue. Essentially, Snowball can be considered a synchronous binary agreement protocol where the agreement property is relaxed to be probabilistic.

In Snowball, a node will repeatedly sample a constant number k of other nodes on the network until it has sufficient confidence that the network is consistent. There are two relevant security parameters – α and β – that are decision-making thresholds employed in the algorithm. Nodes also maintain some internal state for keeping track of their confidence in deciding on a color. Specifically, $d[R]$ and $d[B]$ are counters that are incremented each time a round of sampling results in their respective color receiving a threshold α of votes (where $\alpha > \lfloor \frac{k}{2} \rfloor$). An additional counter, *cnt*, is incremented each time a round of sampling fails to change the node's preference to the other color and is reset to one whenever the color preference switches. For a node to change its color preference, its confidence count in the alternate color must exceed the confidence count in its currently preferred color. That is, if the node currently prefers red, then it will switch to blue only when $d[B] > d[R]$, and vice versa. A node will only consider a decision final on their current color when it has at least β consecutive rounds where the color preference was unchanged (i.e., when $cnt \geq \beta$). More formally, Snowball works as follows in order to decide between red and blue, where a node takes $col_0 \in \{R, B, \perp\}$ as input:

```
1. Initialize state.
    • $col := col_0$
    • $lastcol := col_0$
    • $cnt := 0$
    • $d[R] := 0$
    • $d[B] := 0$
2. While undecided, do:
    (a) If $col = \perp$, continue
    (b) $K := Sample(k)$
    (c) $P := [Query(v, col)$ for $v \in K]$
    (d) $maj := false$
    (e) For $col' \in \{R, B\}$, do:
        i. If $P.COUNT(col') \geq \alpha$, then:
            • $maj := true$
            • $d[col']++$
        ii. If $d[col'] > d[col]$, then: $col := col'$
        iii. If $col' \neq lastcol$, then:
```

- $lastcol := col'$
- $cnt := 1$

iv. Else if $col' = lastcol$, then: $cnt++$

v. If $cnt \geq \beta$ then ACCEPT(col').

(f) If $maj = false$, then: $cnt := 0$

Snowball is able to provide probabilistic consistency guarantees with failure probability ε (not unlike Nakamoto Consensus). Even if the network begins in a fully bivalent state (equal preference for each color), the random perturbations that occur due to sampling will cause the preferences to drift toward one color or the other by some amount δ. As δ increases, it becomes exponentially less likely that the minority value will overtake the majority. Consistency requires properly setting the parameters k, α, and β. The liveness guarantee provided by Snowball is such that if the number of adversarial nodes $f \leq O(\sqrt{n})$, then the protocol terminates with probability $\geq 1 - \varepsilon$ in $O(log\ n)$ rounds. The number of rounds increases as the adversary controls more nodes, requiring an exponentially increasing number as f approaches $\frac{n}{2}$ (as with Nakamoto Consensus).

Avalanche uses Snowball internally to decide between *conflict sets*, or transactions that spend the same funds. When there are no double-spends on the network, a conflict set contains just a single transaction. When clients issue transactions, they specify one or more parent transactions which form the edges of a DAG. The DAG improves efficiency because voting for a particular transaction also counts as a vote on all of its ancestor transactions back to genesis.

Let tx be a transaction that an Avalanche node is trying to decide on. The node will query a sample of its peers to ask about tx. Other nodes will only support tx if every transaction reachable from tx in the DAG is the preferred transaction in their respective conflict sets. Stated differently, a node will only vote yes on tx if $\forall tx_j \in ConflictSet(tx_i)$, $\forall tx_i \in past(tx) \cup \{tx\}$, tx_i is preferred over tx_j. Should a threshold α of yes votes be received, the transaction collects a *chit*. A node's confidence in a transaction is the number of chits in the future set of that transaction, or transactions from which it is reachable in the DAG. As such, confidence increases as the DAG expands. Ties are broken in favor of the transaction that a node saw first. The transaction recipient decides when to accept the transaction based on a threshold of β consecutive chits in its favor.

This procedure is similar to that employed in the Tangle algorithm (Section 11.5.3). A crucial difference is that in Avalanche, confidence in a transaction is not based solely on the structure of the DAG but rather the accumulation of chits. This makes Avalanche resistant to certain attacks that Tangle-based systems are subject to, like parasite chain attacks, where the adversary creates large subDAGs to overpower the honest portion of the graph. Another protocol that is very similar to Avalanche is Fast Probabilistic Consensus (FPC) [356]. FPC assumes that nodes have access to some form of trusted randomness beacon, perhaps through a distributed random number generation process. A round of the FPC protocol works in three steps with a different k queries to different nodes in each round. Replicas maintain a counter that is incremented in each round where the consensus value remains the same, and the protocol terminates for the node when this exceeds another threshold value. The three steps in a round are:

1. Each node samples the beliefs of k other nodes.
2. After sampling, the nodes run some kind of distributed randomness generation protocol to determine a (moving) threshold X_r where $0.5 < X_r < 1$.
3. Each node then chooses 1 if the number of nodes in their sample was greater than $k * X_r$ and 0 otherwise.

The first round has a separate initial threshold value for step two, but what is described above is for all other rounds. Because the random value is chosen *after* the sampling is performed, even a nearly omniscient adversary (knowing everything except the future random value) will not know which threshold to attempt to sway the sample toward. In particular, this helps with speedy termination relative to Snowball when an adversary attempts to interfere. The security threshold for FPC is atypical and approximately 38%, but the reasons are beyond the scope of this document.

13.3.3. Parallel Chains

The idea of composing a ledger out of m separate ledgers was discussed in the proof-of-work context in Section 11.4. In that case, it was used to reduce the settlement latency of transactions. A similar approach for proof of stake was proposed in [357], where the composition of m chains is used to achieve nearly optimal transaction throughput. Unlike the proof-of-work case, this scales throughput rather than latency, and transactions can only appear on a single chain instead of multiple chains. Each chain shares a common genesis block, and the leader election procedure is executed independently for each chain but using the shared stake distribution across the entire system.

The idea from [357] can be used with other chain-based proof-of-stake protocols but is described as the composition of m separate chains that operate using a modified version of Ouroboros Praos (Section 13.1.3) that also adopts the Inclusive rule described in Section 11.5.1. The use of the Inclusive rule means that this scheme runs m parallel DAGs, though a single linear chain is formed for each of the m DAGs by including blocks that would have otherwise been rejected by the longest chain rule. A combining procedure is used to take these m separate DAGs and form a single linearized ledger of transactions. Unlike many DAG protocols, this approach keeps separate graphs that grow more slowly and fork infrequently. Typical DAGs increase throughput by producing blocks rapidly and accepting frequent forks.

Validators include references in their blocks to other blocks off of the main chain other than the specific parent reference, as is typical in DAGs. For a main chain $C = B_1, B_2, ..., B_L$ of blocks B_i, the Inclusive rule includes blocks outside of C into a single chain as follows: for $B_i \in C$, insert before B_i all blocks in $past(B_i) \setminus past(B_{i-1})$, where those blocks are sorted topologically and ties are broken via block hash.

The parallel chains approach is to then perform a comparable procedure across all m chains, $C_1, ..., C_m$. All blocks in all chains are first put in order based on their respective slot index with ties broken based on their chain number $c \in \{1, ..., m\}$. The result is a sequence of blocks $B_1, B_2, ..., B_L$. Then, $\forall i \in \{1, ..., L\}$, insert prior to B_i all blocks from $\bigcup_{i \in \{1,...,m\}} S_i \cap \{past(B_i) \setminus past(B_{i-1})\}$ sorted topologically (and breaking ties by block hash). Here, S_i is the portion of chain C_i that is stable (i.e. , with blocks at the end chopped off to maintain a common prefix). Finally, based on this block ordering, transactions must be sanitized in order to remove those that are invalid.

13.4. BFT-Based Proof of Stake

13.4.1. Tendermint

The Tendermint algorithm, which is an adaptation of the PBFT algorithm to proof of stake, is described in [358]. The original version had flaws that were found and fixed in [359], and the latest version was analyzed and proven secure in [360]. The novelties of Tendermint as compared to PBFT are the use of gossip for networking and the elimination of the separate view change algorithm (similar to some of the algorithms from Section 5). A new leader is elected in each round as part of the normal processing rather than using the additional subprotocol described in Section 4.1. This improves Tendermint's worst-case communication complexity from PBFT's $O(n^4)$ to $O(n^3)$ because processes locally keep track of potentially decided values rather than exchanging all of the messages they have already delivered. Instead of requiring $\frac{2n}{3}$ validator signatures, as in PBFT, Tendermint requires $\frac{2}{3}$ of the total stake to sign off on blocks. For ease of exposition, this will sometimes be described as seeing $2f+1$ signatures or messages. Unlike most other proof-of-stake algorithms, Tendermint's reliance on PBFT makes it secure under partial synchrony rather than only in synchronous networks. Unlike PBFT, validators are not necessarily connected over a complete network but rather communicate via gossip over a peer-to-peer overlay network.

Leader election follows a stake-weighted round-robin process, such that block proposers are selected proportionally to their stake. In Tendermint, a single block is created at a given chain height, but multiple rounds may be required to generate this block if replicas cannot agree on a value under a given leader. Each consensus instance is described by a chain height and round number, and all messages exchanged will include those values. Tendermint uses PROPOSAL, PREVOTE, and PRECOMMIT messages (which correspond to their PBFT equivalents of PRE-PROPOSE, PROPOSE, and COMMIT, respectively), and each of these message phases has a corresponding timeout.

The PROPOSAL message includes the full block proposed, but the PREVOTE and PRECOMMITs just carry a block hash in order to save bandwidth. After receiving a PROPOSAL for value *v*, validators send a PREVOTE for it if valid (or a PREVOTE with a special *nil* value if invalid or timed out). Upon receiving PREVOTE messages for *v* signed by keys associated with more than $\frac{2}{3}$ of the system's stake, validators send a PRECOMMIT message for *v* (or PRECOMMIT with a *nil* value on timeout or if the replica has not seen a valid block with the given hash). Similarly, after receiving $2f+1$ matching PRECOMMITs in a given round, a correct process decides on that value. If multiple rounds are necessary at a given chain height, a correct round leader will propose the same value as the prior round leader if that value is valid. So far, this is just standard PBFT, but Tendermint also includes several locking variables in order to maintain safety across different rounds without a view change mechanism:

- *lockedValue* is the most recent value with respect to a round number for which a PRECOMMIT has been sent.
- *lockedRound* is the last round in which the validator sent a PRECOMMIT that is not *nil*.
- *validValue* is the most recent possible decision value. *validValue* is the last value that a validator delivered $2f+1$ times and can differ from *lockedValue*.
- *validRound* is the last round in which *validValue* was updated.

Leaders send PROPOSAL messages that include their local *validValue* if their $validValue \neq nil$. These messages include *validRound*, so other processes are informed about the last round in which the proposer observed *validValue* as a possible decision value. Other validators then check the PROPOSAL against their own local *lockedValue*. A PROPOSAL is accepted if it is valid and either the round number in the PROPOSAL $\geq lockedRound$ or the PROPOSAL matches their *lockedValue*. Otherwise, they send a PREVOTE with *nil* value, which they also do if there is a timeout and they have yet to send a PREVOTE for the current round. This is *timeoutPropose*, which is triggered when a new round begins.

After receiving the $2f+1$ PREVOTEs on a valid PROPOSAL, a validator "locks" on that value (they set *lockedValue* and *lockedRound* before sending a PRECOMMIT if still in the prevote step and set *validValue* and *validRound* if in the prevote or precommit steps). Otherwise they send a PRECOMMIT to *nil*, which is also sent if a timer expires without having sent a PRECOMMIT in their current round. This is *timeoutPrevote*, and its timer begins when an honest validator sends a PREVOTE or receives $2f+1$ PREVOTEs in a round. The block is committed if the replica sees a valid PROPOSAL and $2f+1$ matching PRECOMMITs, and validators move on to the next height of the chain. At any point, if a validator sees $f+1$ messages of the same kind for a round greater than the local round, the validator advances to that round's proposal step.

Processes "unlock" a block only when the block is committed or when seeing $2f+1$ PRE-

VOTEs on a conflicting block (note that a vulnerability, discussed below, existed at this step until corrected in [359]). When locked on a value, an honest validator only sends messages for that value. There is also a timer to prevent blocking on the final precommit step, *timeoutPrecommit*, where the timer begins upon receiving any $2f+1$ PRECOMMIT messages in the current round, regardless of whether the values conflict. If it expires, they move to the next round.

The original Tendermint protocol had a potential safety violation during the unlocking process. Say that a validator is locked on a block B. If that validator sees $2f+1$ support for block B', they set $lockedValue = nil$. However, the validator must ensure that $B \neq B'$ at this step so that the following scenario does not occur: the validator locks on B in round r and then sees $2f+1$ support for B again in round $r' > r$, which then unlocks B but fails to lock it again. In this case, some validators may commit to B, while others commit to $B' \neq B$.

Tendermint is unable to have light clients work the "standard" way that clients would work for PBFT. Clients do not query validators directly and, thus, cannot check for $f+1$ matching answers. Furthermore, the proof-of-stake validator set is constantly changing, and a light client needs an accurate view of this set in order to check the validity of signatures on blocks. An approach to creating Tendermint light clients was described in [361].

A variant of the Tendermint algorithm, Tenderbake, was designed to be secure even with bounded message buffers [138]. As it stands, validators need to store messages for a potentially unbounded number of rounds. Tenderbake has the additional advantage of terminating more quickly than Tendermint, but it loses the optimistic responsiveness that comes with partial synchrony.

13.4.2. Algorand

Algorand uses proof of stake and *cryptographic sortition* (using a VRF) to elect a committee of validators who then execute a novel single-shot Byzantine agreement algorithm called BA$\star$ [362]. A primary benefit of the algorithm is that forks are (effectively) impossible and, thus, transactions are finalized (almost) immediately upon inclusion in a block. Algorand's liveness relies on a "strong synchrony" assumption, where most honest users will see the messages of most other honest users within a known time bound. Safety requires "weak synchrony," where the network can be asynchronous for a lengthy but bounded interval, but must then be followed by a sufficiently long synchronous period (confusingly, this is distinct from the weakly synchronous model introduced in Section 1.5). A variant of the Algorand protocol can also be modified to be secure in the sleepy model, though with a performance penalty [363].

Algorand can scale to a large number of validators by running each step of BA$\star$ with a committee of only a subset of validators large enough to ensure that no selected committee would exceed the security threshold over the lifetime of the system (with high probability). Messages from the BA$\star$ execution are gossiped across the network instead of sent point-to-point between elected committee members. The BA$\star$ algorithm does not require participants to maintain any private state other than their private keys, and participants are expected to only send a single message, which allows them to be immediately replaced after sending their message. Each step of BA$\star$ has a new set of committee members. Combined with the VRF for sortition, this makes the system adaptively secure and resistant to targeted denial of service on committee members.

The sortition process provides each selected member with a *priority*, which can be compared between participants. Since multiple validators may be selected to propose blocks, priority determines which one should be adopted. Validators initialize BA$\star$ with the highest priority block they have seen, and BA$\star$ then executes in repeated steps. Each step begins with sortition to determine the committee members, and elected members broadcast a proof of their selection over the network. These steps are then repeated until there are enough participants in the committee to reach consensus. Each validator checks whether they were elected for the next step as soon as the previous step ends.

The sortition process uses a *role* parameter that specifies the particular roles that a participant may play in the consensus process (e.g., block producer, committee member in step two, etc.). For any given role, an expected number τ of participants are selected. A single user may be elected as multiple sub-users for a given role based on their stake (i.e., an entity with a large stake may count as multiple members of a single committee). Let *seed* be a random seed to be discussed shortly, w be the number of atomic units of cryptocurrency owned by a particular user, W be the total stake in the system, $p = \frac{\tau}{W}$ be the probability of a particular unit of stake being selected, and the probability that exactly k of the user's w stake units are selected is binomially distributed and denoted as $B(k;w,p)$. Users compute $(y,\pi) = VRF_{sk}(seed||role)$ and then use the pseudorandom value y to determine how many sub-users they control that were elected. To do so, the interval [0,1) is divided into subintervals $I^j = [\sum_{k=0}^{j} B(k;w,p), \sum_{k=0}^{j+1} B(k;w,p))$ for $j \in \{0,1,...,w\}$. If y_{len} is the length of the output of the VRF, then a user has j sub-users selected if the value $\frac{y}{2^{y_{len}}} \in I^j$. The priority of a block produced by sub-user i for $i \in \{1,...,j\}$ is equal to $H(y||i)$.

In Algorand, a new block is appended to the blockchain in each round, and each round r has a corresponding random seed, $seed_r$. When selected to be a block proposer in round $r-1$, validator i also proposes a seed to be used for round r as $(seed_r, \pi) = VRF_{sk_i}(seed_{r-1}||r)$ and includes the seed in their proposed block. Once there is agreement on a block for round $r-1$, all validators will agree on the seed to be used for round r. If a proposed block does not contain a valid seed (or the block is otherwise invalid), participants treat the block as though it were empty and compute $seed_r = H(seed_{r-1}||r)$ for some hash function H.

To be secure, sk_i must be chosen sufficiently far in advance of its use in determining a seed, such that the seed will remain pseudorandom even if i is malicious. To prevent the adversary from manipulating the sortition process, the seed used for sortition itself is changed only every R rounds, such that the round r seed used in sortition is actually $seed_{r-1-(r \bmod R)}$. For some timing parameter b, round r's sortition uses the sk_i's and their associated stake from the last block created b-time before block $r-1-(r \; mod \; R)$. Security requires that there be at least one honestly produced block in this b-timed interval. By keeping b large, network adversaries are less likely to be able to keep the network partitioned long enough to produce all empty blocks and control the seed. On the other hand, a longer b leads to higher stake shift and correspondingly lowers the security margin.

Algorand selects up to $\tau_{proposer}$ proposers via sortition, so multiple blocks may be produced in a round. Validators wait for some short period of time (on the order of 10 seconds) to see blocks in a round before beginning BA$\star$ – keeping the highest priority block they have seen in a round – and ignore blocks with a lower priority. When the time is up, they start BA$\star$ with the highest priority block as input or an empty block if none were seen.

BA$\star$ has two phases, the first of which has two steps, and the second of which has at least two steps. In the first phase, one of the proposed blocks is chosen to compete against an empty block in the second phase, which runs binary Byzantine agreement. Each step has an expected number of participants τ_{step}, except for the final step which has an expected τ_{final}. The voting threshold at each step is then either $T * \tau_{step}$ or $T * \tau_{final}$, where $T > \frac{2}{3}$ is the same at all steps except potentially the final step.

The $CommitteeVote()$ procedure runs sortition for the "committee" role of a given round and step and then votes on a block hash taken as input. The $CountVotes()$ procedure takes the votes for the current round and step and outputs the block hash of the first proposed block to gather the appropriate threshold of votes. At each step, validators run $CommitteeVote()$ on some block hash, and everyone waits for a specified amount of time to count votes before timing out if enough votes are not accumulated.

In the first step of the first phase, committee members vote for the block hash that BA$\star$ was initially passed. In the second step, members vote for either a block hash that received at least $T * \tau_{step}$ or an empty block if the voting threshold is not reached. At the end of this phase, honest participants will have no more than a single non-empty block to consider in the next phase. An empty block is more likely to be chosen if the highest-priority block proposer was malicious and equivocated or if the network is not synchronous.

In the second phase, a binary BA algorithm is executed in order to have validators agree on a choice between an empty block and the hopefully non-empty block output from the first phase. Should a validator receive the threshold of votes needed for a block hash in any step of this phase, they will vote for that block hash if they are elected as a committee member for the next step. On the other hand, the network may not be synchronous, or an adversary may allow a particular validator to see enough votes to decide on a block while preventing other validators from seeing enough votes before they time out. In this case, the algorithm must ensure that votes in the next step do not result in a different block being decided by different validators, so members' next step votes are of a specific value that could have been returned in a given step, and each step can only return one particular value. In addition, whenever a participant returns a block from the protocol, they continue to vote for it for the following three steps in order to help that value gain enough votes in future steps. Specifically, given a parameter $MAXSTEPS$, the binary Byzantine agreement protocol works as follows while $step < MAXSTEPS$:

```
1. Check for agreement on $block\_hash$:
    (a) $CommitteeVote()$
    (b) $v \leftarrow CountVotes()$
    (c) If $v = TIMEOUT$, then $v \leftarrow block\_hash$
    (d) Else if $v \neq empty\_hash$:
        i. For $step < s' \leq step + 3$, call $CommitteeVote()$
        ii. If $step = 1$, call $CommitteeVote(FINAL)$
        iii. Return $v$
    (e) $step{+}{+}$
2. Check for agreement on $empty\_hash$:
    (a) $CommitteeVote()$
    (b) $v \leftarrow CountVotes()$
    (c) If $v = TIMEOUT$, then $v \leftarrow empty\_hash$
    (d) Else if $v = empty\_hash$:
        i. For $step < s' \leq step + 3$, call $CommitteeVote()$
        ii. Return $v$
    (e) $step{+}{+}$
3. Use a common coin:
    (a) $CommitteeVote()$
    (b) $v \leftarrow CountVotes()$
    (c) If $v = TIMEOUT$ and $CommonCoin() = 0$, then $v = block\_hash$
    (d) If $v = TIMEOUT$ and $CommonCoin() = 1$, then $v = empty\_hash$
    (e) $step{+}{+}$
```

With an honest block proposer and a synchronous network, most committee members will begin with the same block and return on the first step. However, during a network partition, it is possible for honest users to return different blocks in the protocol as described so far.

Algorand solves this through the notions of *tentative consensus* and *final consensus*. After the binary BA algorithm returns, there is one last voting step in BA$\star$ in which votes are counted and a block is decided if validators see $T * \tau_{final}$ votes for it. Final consensus exists if enough validators return at the first step and then see enough votes from other committee members demonstrating that they did too. If the highest priority block proposer was honest and the network is synchronous, then final consensus is achieved after four interactive steps. A particularly lucky adversary may cause this to take an expected 13 steps, which requires the adversary to be the highest priority proposer for the round and to control a large fraction of committee members at each step.

An adversary who controls the network can prevent a subset of validators from reaching either final or tentative consensus for an arbitrary number of steps, and each step increases the adversary's chance of getting consensus on an empty block. To prevent this, BA$\star$ has a bounded number of steps, $MAXSTEPS$, before halting and requiring a recovery procedure. For an attacker to keep the validators from agreeing, they need to know how a participant will vote after they have a timeout during vote counting. The $CommonCoin()$ procedure makes this more difficult. Each user sets their common coin to be the least significant bit of the lowest committee member hash (produced from the VRF) seen during the step.

When consensus on a block is only tentative, it is possible for multiple forks to exist, which can inhibit liveness. To remedy this, users passively monitor all BA$\star$ votes to keep track of all forks and periodically use sortition to propose one of the forks for finalization. If a validator is chosen via sortition, they propose an empty block that extends their preferred fork of the chain. Other validators then wait for the highest priority proposal, check that it extends their longest local chain, and invoke BA$\star$ on that proposed block.

Finally, Algorand's design does not require checkpointing or having parties regularly be online. The BA$\star$ process results in a *certificate* that can be used to prove consensus on a block. Specifically, a certificate is composed of the votes from the last step of the binary BA algorithm (but not the "final" step that comes after).

14. Hybrid and Alternative Sybil-Resistance Mechanisms

14.1. Proof of Space

While proof of work and proof of stake get most of the attention when it comes to Sybil-resistance mechanisms in permissionless environments, another promising avenue is *proof of space* (PoSpace) [364]. In a proof-of-space system, users must prove that they are dedicating disk space rather than computational resources through a challenge that requires a significant amount of memory or disk space to solve.

The proof-of-space protocol introduced in [364] is an interactive protocol between a prover and a verifier that has an initialization phase and an execution phase. The initialization phase results in the prover storing some data and the verifier storing a short commitment to the data. In the execution phase, the verifier sends a challenge to the prover, who responds with an answer that requires reading some portion of the original data stored on disk. In a consensus setting, there is no designated verifier, so anyone with a view of the public ledger must be able to verify the proof.

The security ramifications of proof of space may be similar to that of ASIC-resistant proof of work in some ways (see Section 9.1.2). Because hard drives are widely available and distributed, proof of space allows people with typical consumer-grade hardware to participate in consensus using their idle resources. Further, storage is never "used up," so hard drives used for proving storage can be used for useful storage later. Disk space can be repurposed and thus has high salvage value, like a CPU or GPU that can be used for other purposes when not participating in consensus. On the other hand, proof of space requires few ongoing costs, unlike proof of work. The cost of participation is the upfront cost of acquiring storage plus the opportunity costs of using that storage to participate in consensus instead of doing something else with it. Stated differently, the vast majority of the expenses incurred from proof of space are capital expenditures rather than operational expenditures. In this respect, proof of space is more similar to proof of stake.

14.1.1. Spacemint

Spacemint is an academic proof of concept of a cryptocurrency that uses proof of space instead of proof of work [365]. Adapting proof of space to the cryptocurrency setting requires addressing several challenges, including two that are familiar from proof of stake: grinding attacks and costless simulation to mine multiple chains simultaneously (see Section 12.1.1).

In a grinding attack, an adversary who mines block B_i can (cheaply) try out multiple block hashes and use one that gives them an advantage in constructing block B_{i+1}. Spacemint decouples the hash chain that includes the proofs of space from a separate signature chain for transactions, which prevents adversaries from manipulating the block hash based on the included transactions. The transactions must be bound to the proof chain, so the purpose of the signature chain is to prevent past transactions from being altered when new blocks are added. This architecture is displayed in Figure 42.

Spacemint addresses the costless simulation issue by defining a proof's *quality* and making it fixed for a given time step in order to eliminate the benefit of trying many chains. The probability that a proof has the best quality should be equal to that miner's fraction of the total space being used to mine the chain. Miners who publish multiple conflicting blocks can then be punished by having their reward forfeit and giving half of it to the miner who includes proof of equivocation. Because the proof-of-space miner does not have stake that can be taken, this punishment is limited and less effective than it is in proof-of-stake systems. The penalty only deters attacks like selfish mining, where the potential gain may be less than the potential penalty, but is unlikely to discourage double-spending attacks.

The proof-of-space challenge for block B_i is the hash of block $B_{i-\delta}$, where δ is parameter-

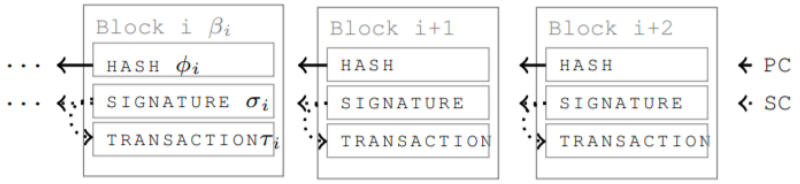


Fig. 42. Spacemint grinding defense. The proof chain is denoted PC, and the signature chain is denoted SC. If an honest miner mines the i-th block and does not equivocate, then past transactions cannot be changed. If an adversary wanted to change transactions in the j-th block, $j < i$, while maintaining the same proof chain, they would need to compute new signatures for all of the blocks in between, which requires the corresponding secret keys. [365]

ized such that $B_{i-\delta}$ is a block that was mined a short while in the past. However, δ cannot be set arbitrarily high because miners can precompute their answers for δ time steps and then not need to access their storage for some time. This would potentially allow a miner to reuse the same storage space for multiple space commitments because they could perform the initialization procedure multiple times. Therefore, a related security requirement is that initialization must be time consuming (at least δ blocks' worth of time).

A final consideration in Spacemint is that miners must commit to the storage space they intend to mine with in advance. This requires a special space commitment transaction that specifies the miner's public key and their space commitment used for verification. If miners were not required to commit this in advance on the blockchain, then they would be able to reuse the same space for multiple commitments due to properties of the proof-of-space scheme from [364]. An unfortunate side-effect of this is that if an adversary acquired the majority of the storage space used to secure the chain, they could censor new space commitment transactions and maintain their majority indefinitely.

14.1.2. Chia

The Chia network consensus algorithm is inspired by Spacemint, but adds a verifiable delay function (VDF) and fixes some of its weaknesses. The description in this section corresponds to the original Chia "Green Paper" [366], but the Chia network as deployed may differ slightly from the presentation here.

Chia addresses grinding attacks by combining two interlinked chains in a manner similar to Spacemint; an ungrindable *trunk* contains the proof-of-space and VDF output, while the *foliage* chain contains transactions, timestamps, and signatures that bind the foliage to the trunk. The proof-of-space challenge is derived from the last VDF output. Specifically, the challenge is the hash of a BLS signature over the VDF output, which is unpredictable to an adversary. The uniqueness of BLS signatures, where there is only one valid signature for a given public key and message pair, prevents grinding over different challenges in the trunk. This architecture is shown in Figure 43.

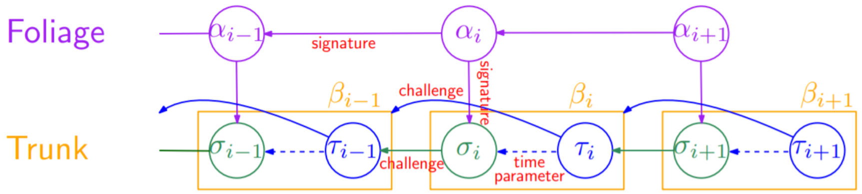


Fig. 43. Chia design. A block $\beta = (\sigma, \tau)$ – where σ is a proof of space and τ is a VDF output – is in the ungrindable trunk chain. The foliage blocks α_j contain transactions, a signature on the previous foliage block (to create a blockchain linked by signatures), and a signature on the proof of space (to bind the foliage to the trunk). Because the challenges for the proof of space and VDF come from previous trunk values, grinding on various values of data in the foliage does not give the attacker an advantage. [366]

Chia uses a proof-of-space algorithm where the initialization procedure is non-interactive, unlike the one from [364] used in Spacemint. This removes the need to pre-register the storage commitment in a special transaction, which Spacemint uses to prevent grinding attacks.

Chia alternates between proofs of space and VDF executions, where the VDF is used to protect against long-range attacks and remove the need for synchronized clocks. In Chia parlance, the entities that compute the proofs of space are called *farmers*, and the ones that compute the VDF are called *time lords*. In a proof-of-space system, unlike proof of work or proof of stake, any farmer can generate a valid proof during any time slot, so a mechanism is required to determine which block would "win." To prevent unbounded bandwidth consumption, proofs that are unlikely to win are disallowed in the first place. Chia adopts the idea of assigning a quality to each proof of space (proportional to the amount of space used to construct it) and agreeing on the best quality one as the legitimate chain extension, as was done in Spacemint. Here, quality is the hash of the proof, and the best quality proof is the one with the lowest valued hash. This mechanism is augmented by requiring a VDF output in order to consider the block valid. The VDF is parameterized such that the time it takes the time lords to compute it is linear in the quality of the proof of space (specifically, the VDF time parameter is the quality times the current difficulty level), which leads to having the best quality proofs completed first, as desired.

The use of the VDF opens up the possibility for another subtle grinding vector that could be used to launch double-spending attacks, which is addressed by slightly modifying the difficulty adjustment from the one used by Bitcoin. The time parameter for the VDF depends on the difficulty level, which is computed from the timestamps located in the foliage. Grinding on the timestamp in the foliage can lead to grinding against the VDF output in the trunk if the new difficulty kicks in immediately following the block whose timestamp was used to recompute the difficulty, as is done in Bitcoin. This threat is prevented by having the difficulty adjust only after several additional blocks have been constructed. An example of this grinding attack is shown in Figure 44.

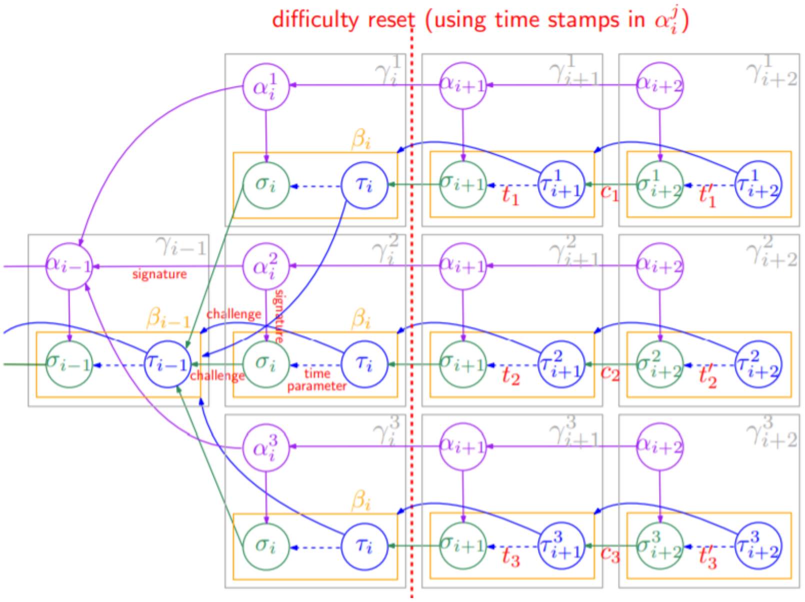


Fig. 44. Chia difficulty grinding. The foliage blocks α_i^1, α_i^2, and α_i^3 each have different timestamps. Because they occur right before the difficulty reset at block i, each of the three chains has a slightly different difficulty parameter to be used for blocks at height $i+1$ or greater. The adversary can run three VDF executions in parallel, and there will be three different outputs at height $i+1$, causing each chain to have a different proof-of-space challenge, c_1, c_2, or c_3 for blocks at height $i+2$. [366]

Chia does not follow Spacemint in using punishment to discourage costless simulation attacks where farmers extend multiple chains at once. Instead, it embraces this "double-dipping" and allows honest farmers to engage in the practice as well. If honest farmers do not double-dip, then at least $\approx 73.1\%$ of the total allocated space must be supplied by honest farmers. Chia improves this bound by adding a local security convention where honest farmers extend the first $\kappa > 1$ chains at every chain depth. It is recommended that $\kappa = 3$, which reduces the security margin to requiring $\approx 61.5\%$ of space to be supplied by honest farmers. A higher value of κ improves the security margin (Chia would be secure with an honest majority if farmers were required to extend every block) but requires that honest farmers compute a factor of κ more proofs and makes agreement take longer.

While the chain quality and chain growth properties were demonstrated in the original Chia paper, the crucial common prefix property was not proven until [225]. Chia's security analysis differs from that of chain-based proof of stake, despite both suffering from a variant of the nothing-at-stake problem. This difference arises because in, say, Ouroboros Praos and Snow White, the same randomness is used across multiple blocks, whereas Chia's randomness is independent at each block. This gives an adversary in Chia an independent opportunity for winning leader election at every block, which magnifies the attacker's power by a factor of e.

If one were to naively adopt an equivalent of a longest chain rule and apply it to a proof-of-space system, one might have honest farmers agree on the chain where the sum of the qualities of the included proofs is highest. This opens up the possibility of long-range attacks because it is possible to generate a heavier chain as long as the adversary has enough space to beat the *average* of the space used across the entire honest chain. If the value of farming Chia blocks increased significantly or the cost of storage decreased significantly after the chain began, beating the average might be much easier than overpowering the amount of space currently allocated to Chia farming. Spacemint mitigates this by only considering the quality of more recent blocks when comparing the quality of chains. This is not a problem for Chia because the VDF prevents an adversary who forks the chain far in the past from catching up to the honest chain.

14.2. Proof of Activity

Proof of activity (PoA) is a hybrid proof-of-work and proof-of-stake system intended to diversify the types of entities securing the blockchain, particularly as transaction fees become a more important component of the block reward [367]. Stakers pick transactions to include in blocks, which makes it harder for miners to censor transactions using feather-forking or majority attacks. PoA requires that a majority of the online stake is honest in order to maintain security.

Leader election is performed using the "follow-the-satoshi" idea, similar to how it is done in the Chains of Activity protocol (Section 13.1.1). A pseudorandom seed is used to select uniformly from the total set of *satoshis* (atomic cryptocurrency units) based on when it was minted. That satoshi is then traced through the transaction graph to the public key that currently controls it, which determines the leader. Note that this procedure can cause some challenges when using more advanced scripting capabilities than simple public key to public key financial transfers.

The PoA protocol begins by having miners mine a block header that does not include any transactions and broadcasting the empty block to the network upon success. The hash of this block header is then used as the seed to deterministically derive N pseudorandom stakeholders (with $N = 3$ suggested). The follow-the-satoshi procedure is invoked N times using $H(header\_hash||previous\_block\_hash||i)$ for $i \in \{1,...,N\}$. All online stakeholders check whether they are one of the N selected stakeholders for that block. The first $N-1$ stakeholders sign the hash of the empty header and broadcast it to the network. The N-th stakeholder creates and broadcasts a *wrapped block* that extends the empty header by including transactions, the $N-1$ other signatures, and their own signature over all of this information. Nodes continue to follow the longest chain rule, as in Nakamoto Consensus. The transaction fees are shared between the block's miner and the N stakeholders.

If not all of the N stakeholders are online, another miner will mine another header at the same height of the chain, which will derive a different N stakeholders. The use of proof of work limits the ability of an attacker to bias the randomness used for the stake election. To incentivize a certain level of online stake, the N-th stakeholder can include in their block all of the other empty blocks mined at the same height but without the required signatures. Nodes can count how many empty blocks were mined during a difficulty retargeting window. If too many were mined, then stakeholders can get a higher fraction of the block reward during the next window, and vice versa.

By combining PoW and PoS, an adversary may need to acquire both a significant amount of computational power *and* stake to pull off attacks. Specifically, if the follow-the-satoshi procedure is a random oracle, then an attacker with α fraction of the online stake needs more than $(\frac{1}{\alpha}-1)^N$ times the hash power of honest miners in order to perform typical majority attacks, like censorship and double-spending. An implication of this is that an adversary with the majority of the online stake will actually have their computational power magnified substantially. On the other hand, with $N = 3$ and only one-third of the online stake, an adversary requires eight times more computational power than the honest nodes. The inclusion of proof of stake increases the susceptibility of PoA to bribery attacks, which become more severe as N increases. An attacker may be able to bribe stakeholders to withhold their signatures and only needs to bribe a single stakeholder per block.

14.3. Checkpoints and Finality Gadgets

The *CAP theorem* states that a distributed system can only maintain consistency or availability in the face of a network partition but not both. Many of the protocols discussed in this document, most notably Nakamoto Consensus, favor availability over consistency. When the network of miners is partitioned, transactions can still be put into blocks on both sides of the partition (availability), but these transactions may conflict and need to be reconciled when the partition ends. Eventually, the conflict is resolved, so Nakamoto Consensus has *eventual consistency*. However, even this notion of eventual consistency leaves a small possibility that a block may be reverted in the future.

Contrast this with the idea of *finality*. Permissioned systems, some committee-based proof-of-work systems, and BFT-based proof of stake tend to have finality, where a finalized block can never be reverted under any circumstances. As a trade-off, this means that the system halts during a network partition. Finality is a desirable property because any application that relies on the underlying state machine can truly treat a finalized transaction as final rather than needing to be prepared to revert its execution. Furthermore, it is not too difficult to notice when the system halts for an extended period, so manual intervention may be able to fix the availability issues fairly quickly. This section discusses protocols for adding finality to permissionless blockchain systems, including those that otherwise favor availability.

14.3.1. Ad Hoc Finality Layers and Reorg Protection

The earliest form of ad hoc checkpointing were the hard-coded checkpoints that Satoshi included in early versions of the Bitcoin software. Typically, during a software release, Satoshi would pick a block hash from a few months earlier, embed that hash into the node software, and enforce a rule that the node will never accept a chain that does not include the hard-coded checkpoints. While this technically provided finality for transactions through the latest checkpoint block, it was generally too far in the past to be of practical use. The real benefit of hard-coded checkpoints in early Bitcoin was to prevent denial-of-service attacks that stemmed from the very low difficulty of mining blocks at that time. A modestly resourced attacker could generate long chains of low difficulty blocks to fill up a node's disk or prevent them from being able to synchronize the chain. This threat was later solved when the Bitcoin software was updated to perform *headers-first synchronization*, where the node verifies a complete chain of headers and their proofs of work before downloading the contents of the blocks themselves.

Hard-coded checkpoints like this provide finality in a way that is easy to reason about. It also grants software developers considerable centralized power. While this is not much of an issue if the checkpoints are from the distant past, as in early Bitcoin, hard-coded checkpoints are only practically useful for finality if issued for the recent past. In this case, it may allow developers to unfairly determine which chain might be preferred as the canonical one.

Other proof-of-work cryptocurrencies have added more sophisticated forms of finality, or *reorg protection*, primarily to mitigate the threat of majority hash rate attacks. For example, several of the more popular Bitcoin Cash clients include a default rule that they will not perform a reorg of more than 10 blocks at any time, even if the alternative chain has greater proof of work. The justification for this extra rule was that, because only a small minority of the available double-SHA-256 computational power was mining on Bitcoin Cash, the network was highly susceptible to a 51% attack, and cryptocurrency exchanges wanted to be protected from this risk. If the exchange ran a client enforcing this rule, then they could accept deposits after 10 confirmations knowing that even a well-resourced attacker could not revert. As a side benefit, checkpoints also reduce the profitability of selfish mining.

That said, these "moving checkpoints" are not without risk. Most obviously, this rule creates a race condition that can cause accidental chain splits. If the network is partitioned for a few hours, or if an adversary mines a 10-block private chain and publishes it at the right time, there will be a permanent chain split that requires manual intervention to fix. In addition, the new rule removes one of the biggest advantages of Nakamoto Consensus by making the system weakly subjective (see Section 12.1.2), such that a bootstrapping node (or one that has been offline for a few hours) can no longer be assured that the most-work chain is the correct one and must acquire it from a trusted source.

In addition, the finality benefits may be illusory. For instance, an exchange that is temporarily partitioned from the network may have finality for a customer deposit on the particular chain that the exchange is following, but if the network splits, *the finalized chain might not be the one that the exchange wants to follow, particularly if most of the network is following a different chain*. In this case, an exchange that would have quickly rejoined the rest of the network at the conclusion of the attack instead has a false sense of security and must manually intervene in order to join the rest of the network while still suffering from a double-spend attack.

Reorg protection protocols of this nature were formalized in [368], where an additional "front-running" attack was proposed that prevents liveness when there is a rushing adversary with a hash rate majority. The adversary mines their own private chain without adopting any honest blocks. Whenever a new block is published that extends the longest honest chain, the adversary publishes a block while keeping the remainder of their private chain hidden. Because the adversary has the majority of the computational power, they can counter every block, and because they are rushing, adversarial blocks will always win. In this case, every block on the canonical chain will be adversarial. While the intentions of reorg protection assume that there is an adversarial hash rate majority, it is not trivial for an adversary to be rushing on a permissionless network, so this attack is not easy to pull off. The checkpointing mechanisms presented in [368] resist this attack by operating as an unpredictable randomness beacon that "refreshes" the execution at each checkpoint, preventing the adversary from using blocks mined before the checkpoint was issued so that they do not maintain a persistent advantage.

Another interesting implementation of reorg protection is the *ChainLocks* system deployed on the Dash cryptocurrency network [369]. Dash was a pure proof-of-work Nakamoto Consensus cryptocurrency until ChainLocks was added, making it a hybrid with proof of stake. Any Dash user with 1000 Dash can become a *masternode* and provide extra services to the network in exchange for some of the block reward. For each block that is published, a set of a few hundred masternodes is selected, which attempt to finalize the block by issuing a ChainLock on it, where a ChainLock is a BLS threshold signature. Each selected masternode submits a signature share for the first block that they see at each chain height, and if enough of these masternodes sign the same block, the threshold signature is created and broadcast to the rest of the network as a ChainLock. Any node that has seen a ChainLock for a given block will not reorg past that block. Under normal network conditions, this will likely result in blocks being finalized almost immediately and make it more difficult for an adversary to cause a chain split. This also gives the masternodes a significant amount of power that can be wielded to an adversary's advantage, but the adversary would need to own a very large amount of Dash (well over half) or be able to bribe other masternode owners in order to take advantage of it.

14.3.2. Casper the Friendly Finality Gadget (FFG)

The Ethereum network began as a pure proof-of-work cryptocurrency that followed the GHOST fork-choice rule but is currently undergoing a multi-phase process to become a pure proof-of-stake network colloquially called Ethereum 2.0, as described in Section 13.2. One step of the process is to adopt a hybrid approach of maintaining the underlying proof-of-work chain and augmenting it with a proof-of-stake finality gadget called Casper FFG

[370, 371]. The intention is to combine Casper FFG and the GHOST fork-choice rule ("Gasper"), as described in [350], which can work in a pure proof-of-stake system as well.

Casper FFG uses any block proposal mechanism (e.g., leader election via proof of work) and uses proof of stake to finalize blocks, called checkpoints, such that a checkpoint can never be reverted. Checkpoints are created 100 blocks apart and form a checkpoint tree or chain. Casper provides *accountable safety* in that it is impossible for two conflicting checkpoints to be finalized unless more than $\frac{1}{3}$ of the deposited stake violates a *slashing condition* (accountability in BFT protocols is discussed in Section 7.2) and can thus be punished. Casper also provides *plausible liveness*, such that if $\geq \frac{2}{3}$ of the stake is honest, it is possible to continue finalizing new checkpoints without having any validators violate a slashing condition.

To be entitled to vote in Casper FFG, a potential validator must put up a stake deposit as collateral to be slashed in case of misbehavior. A vote consists of a signed message containing hashes of *source* and *target* checkpoints, denoted s and t, as well as the heights of s and t in the checkpoint tree, $h(s)$ and $h(t)$. The target checkpoint must be a descendant of the source in the checkpoint tree. Casper uses the following definitions:

- A *supermajority link* is an ordered pair of checkpoints $a \rightarrow b$ where at least $\frac{2}{3}$ of validators have cast votes with source a and target b. Supermajority links may skip over checkpoint blocks.
- A checkpoint c is considered *justified* if it is the genesis block or if there exists a supermajority link $c' \rightarrow c$ where c' is justified.
- A checkpoint c is considered *finalized* if it is the genesis block or if the following conditions hold: c is justified, there exists a supermajority link $c \rightarrow c'$, there is no conflict between c and c', and $h(c') = h(c) + 1$.
- The *dynasty* of a block b is the number of finalized checkpoints in the chain from the genesis block to the parent of block b.

Justification and finalization are analogous to the first and second voting rounds in a typical BFT protocol (i.e., prepare and commit, respectively, in PBFT). Recall that Casper provides accountable safety, meaning that the only way for two conflicting checkpoints to become finalized is for at least $\frac{1}{3}$ of the deposited stake to violate a slashing condition. If a validator violates one of these conditions, an honest validator can prove it and submit a special transaction to the blockchain that forfeits the malicious validator's deposit and rewards the honest validator with some portion of it. A validator may not publish two distinct votes, $(s_1, t_1, h(s_1), h(t_1))$, and $(s_2, t_2, h(s_2), h(t_2))$ where either of the following is true:

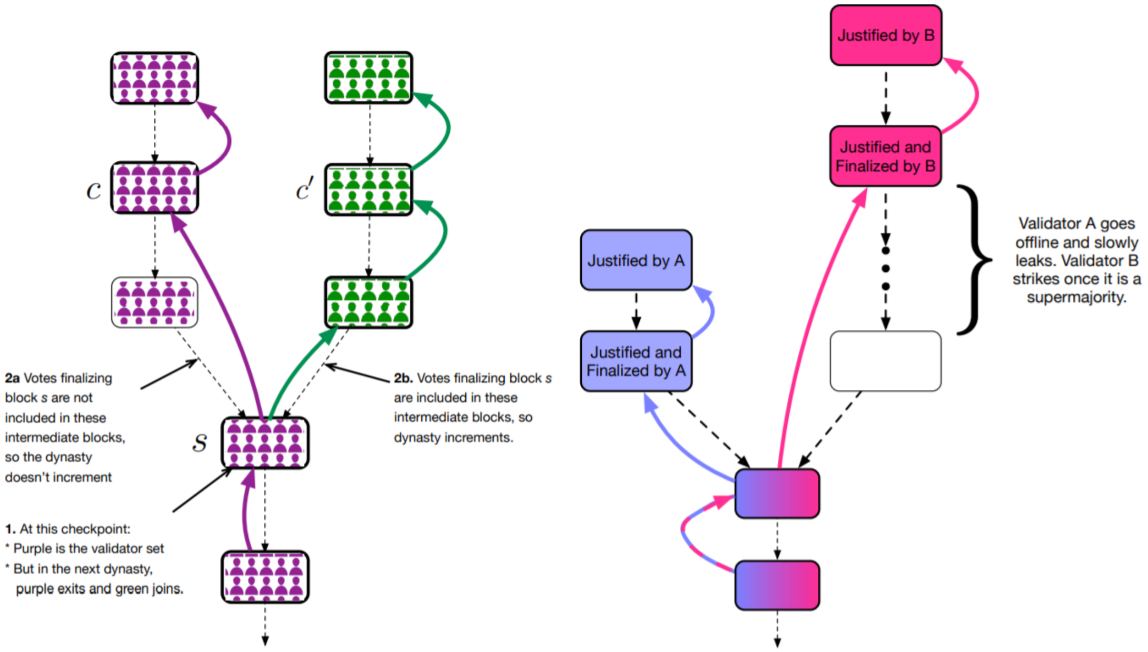
- $h(t_1) = h(t_2)$. That is, a validator may not vote for the same target height twice.
- $h(s_1) < h(s_2) < h(t_2) < h(t_1)$. That is, a validator may not vote within the span of its other votes.

Honest validators in Casper FFG will follow the chain that contains the justified checkpoint of the greatest height and never revert past a known finalized checkpoint. Where ties exist, chains are prioritized based on the underlying proof-of-work scheme, be it Nakamoto Consensus or GHOST.

Slight adjustments must be made in order for Casper FFG to safely handle dynamic validator sets. To join the set of validators, a user sends a deposit message. If it is included in a block with dynasty d, then this validator joins the validator set at the first block with dynasty $d+2$, and this starting dynasty is denoted as $DS(v)$. To exit, withdrawal messages are handled similarly, with the ending dynasty denoted $DE(v)$. When $DE(v)$ begins, the deposit is locked for some period of time so that the exiting validator can be slashed for misbehavior. Define the *forward validator set* and *rear validator set* for dynasty d as $V_f(d) \equiv \{v : DS(v) \leq d < DE(v)\}$ and $V_r(d) \equiv \{v : DS(v) < d \leq DE(v)\}$, respectively. To handle these validator sets, Casper redefines a supermajority link (s,t) for a target in dynasty d such that at least $\frac{2}{3}$ of both $V_f(d)$ and $V_r(d)$ have published votes $s \rightarrow t$. The finalization of checkpoint c then has an additional requirement: the votes for the two supermajority links $c \rightarrow c'$ and the one that justifies c must be included in c''s blockchain prior to the child of c' or before block $h(c') * 100 + 1$. This change prevents safety violations in the pathological case where two grandchild blocks of a finalized checkpoint have different dynasties and evidence of slashing condition violations are included in one chain but not the other. This situation is displayed in Figure 45a.

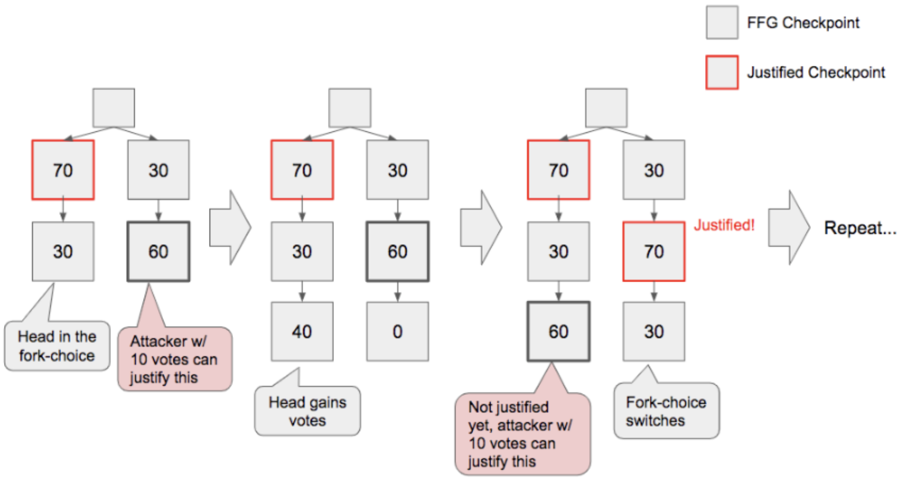
A final component of Casper FFG is the *inactivity leak*, which is intended to maintain liveness even if more than a third of validators are partitioned from the network or crash at the same time. In this case, supermajority links can no longer be created, preventing the finalization of additional checkpoints. For this reason, when validators fail to vote for checkpoints, they are slowly penalized by having their deposit reduced until connected validators become a supermajority again. This mechanism, shown in Figure 45b, can result in conflicting finalized blocks and cause chain splits. In practice, this would require the network partition to last for about three weeks [371].

There is a liveness attack against Casper FFG called the *bouncing attack*, which allows an attacker with less than $\frac{1}{3}$ of the total stake to prevent finalization by "bouncing" between justifying one side of a fork or another [372]. The attack is possible when there is a latest justified checkpoint C and *justifiable* checkpoint C' such that C' is from a later epoch than C and conflicts with C. A justifiable checkpoint is one in which the attacker has enough votes to justify it but has not published them. A potential fix is to only allow the latest justified checkpoint to change during the first third of an epoch, otherwise marking it as pending and reevaluating when the epoch ends. A comparable attack is also possible against the Gasper protocol that combines Casper FFG with GHOST [373, 374]. The bouncing attack is shown in Figure 45c.



(a) Dynamic validator sets

(b) Inactivity leak



(c) Bouncing attack

Fig. 45. Casper FFG attacks. (a) The validator sets finalizing checkpoints c and c' are disjoint, so no one gets slashed despite violating the slashing condition that c and c' are at the same height. (b) The checkpoint on the left can be finalized immediately, but a network partition prevents some validators from seeing the relevant votes. If the partition lasts for enough time, the stake deposits of those voters who support the left checkpoint will continuously deplete until a supermajority link can be formed on the right side. (c) Bouncing attack on Casper FFG. [370, 372]

14.3.3. More Finality Gadgets and Checkpointing Protocols

Several other checkpointing protocols have been proposed [373, 375–379], and this section compares a few at a high level.

Winkle is a very different checkpointing mechanism that was designed to thwart long-range attacks in account-based proof-of-stake systems but can be used as a more general checkpointing mechanism as well [377]. In Winkle, clients issue stake-weighted votes for checkpoints with their transactions by including a hash of the most recent block they are aware of. A checkpoint is issued whenever at least $\frac{2}{3}$ of the total stake has voted in favor of a block. Based on actual transaction data, a checkpoint on Ethereum would take between 50 days and a year to finalize; on Bitcoin, it would take between four months and three years. To speed up the checkpointing process, Winkle also allows stake to be delegated, in which case a new checkpoint could be issued every few hours or days.

Most similar to Casper FFG is a finality gadget called GRANDPA, which is deployed on the Polkadot network [376]. The biggest difference is that participants in GRANDPA vote on the block of the greatest height they are aware of rather than a block at a predetermined chain height, and the vote transitively applies to all blocks preceding the one voted on. Roughly, the highest block with a supermajority of votes becomes finalized. It operates very similarly to PBFT, where there are two rounds of voting to finalize the chain and explicit timeouts to begin a new round. Because it is partially synchronous, finalization stops if there is a network partition, though blocks continue to be produced. Unlike with Casper FFG, there is no inactivity leak to continue finalization during a partition. After the partition ends, finalization only needs to happen once near the chain tip rather than for everything that came before. Like Casper FFG, GRANDPA provides accountable safety and allows for the ability to slash participants' stake for misbehaving.

Finality layers for eventually consistent blockchains were formalized in [375], where the Afgjort finality layer was proposed. In the model provided, a finality layer must:

- Form a chain of finalized blocks;
- Have all parties agree on the finalized blocks;
- Ensure that the last finalized block does not fall too far behind the last block of the underlying block proposal mechanism (i.e., the finalized chain should grow about as fast as the underlying chain); and
- Require that all finalized blocks have at some point been on the chain accepted by at least k honest parties measured as a fraction of stake or computation, depending on the Sybil-resistance mechanism used for the underlying blockchain (k-support).

The Afgjort protocol can speed up finality by an order of magnitude or so compared to eventually consistent algorithms like Nakamoto Consensus under good conditions and can "turn off" to avoid safety violations under bad conditions, such as an adversary trying to disrupt the protocol. In particular, blocks are declared final once they are in the common prefix of the underlying blockchain, but blocks are no longer finalized when there are between $\frac{n}{3}$ and $\frac{n}{2}$ adversarial nodes, though the underlying blockchain remains live.

Naively, a simple way to finalize a block at height d would be to run a typical Byzantine agreement algorithm and consider the agreed upon block as final. Unfortunately, the validity property of most BA algorithms only guarantees that if every honest party starts with the same input, that is the agreed-upon value. However, if honest parties start with different inputs, then the agreed-upon value may be arbitrary. This implies that if the BA algorithm is executed before the common prefix of all honest parties includes height d, then an arbitrary block absent from the chains of all honest parties may become finalized and violate the k-support property.

Afgjort addresses this issue in a rather intuitive way: wait until the same block at height d is in the common prefix of all honest parties. When a member of the finalization committee has a valid chain up to height $d+1$, they vote on the block at height d in some BFT algorithm that requires unanimity to succeed, and the block is finalized if successful. Otherwise, committee members will continue attempting to finalize the block until they do succeed, where the i-th attempt occurs when they have a chain of height $d+2^i$. This exponential backoff guarantees that if the underlying blockchain is secure, d is eventually in the common prefix.

Of course, while all honest parties may have d in their common prefix, adversarial nodes can always vote to prevent unanimity. There are two Afgjort variants that solve this issue. The first is more efficient and secure but requires the additional security assumption of *bounded dishonest chain growth*, which states that a chain only adopted by dishonest parties grows more slowly than the chains of honest parties. This assumption will not hold if the underlying blockchain uses proof of work but has a difficulty adjustment algorithm that adjusts rapidly. If it does hold, however, Afgjort simply requires that participants voting for the block at depth d also send the 2^i following block hashes in order to justify their vote. Participants then run a subprotocol, Freeze, that quickly settles on a uniquely justified block or $\perp$, followed by binary Byzantine agreement to agree on that block. The randomness in the binary agreement algorithm comes from a VRF. The protocol satisfies all desired properties above with $\frac{n}{3}$-support as long as fewer than $\frac{n}{3}$ corrupted parties are in the finalization committee. The alternative protocol that does not rely on the bounded dishonest chain growth assumption adds an extra filtering step to the beginning of the Freeze subprotocol that attempts to remove votes that came from dishonest parties. It does so by ignoring any votes that were supported by fewer than $f+1$ committee members. As a consequence, this version only has 1-support, a much weaker property.

Afgjort must also ensure that finalization does not fall too far behind the underlying chain's growth. To this end, block producers include a pointer to the most recently finalized block and committee member signatures in their block to attest to this finalization. When the chain grows too quickly, there will be blocks produced where the pointer differs from the actual most recently finalized block, and the next block to finalize can be adjusted ahead to account for this.

Afgjort improves upon GRANDPA in a number of ways. GRANDPA only achieves 1-support as opposed to Afgjort's $\frac{n}{3}$-support. GRANDPA also relies on a leader, which may impact liveness if the leader is corrupted or suffers from a denial of service. Finally, GRANDPA includes explicit fixed timeouts, which prevent it from being responsive. Afgjort can finalize blocks based on the actual network delay rather than needing to wait for timeouts to occur.

More recently, a family of so-called *snap-and-chat* protocols, inspired by Flexible BFT (Section 7.5), has been proposed [373, 378]. Snap-and-chat protocols output two ledgers instead of one: a dynamically available ledger that can remain secure with an unknown number of nodes that may go offline temporarily (such as a blockchain using Nakamoto Consensus) and a finalized ledger. When network conditions are poor, the dynamically available ledger is live but potentially unsafe, while the finalized ledger is safe but may not be live. When the network becomes synchronous again, the dynamically available ledger reconciles any inconsistencies while the finalized ledger catches up. Similar to Flexible BFT, snap-and-chat protocols support clients with differing beliefs: clients who prioritize safety under network partitions can follow the finalized chain, while those who prefer availability can follow the other. When the network heals, all clients will agree on a single history. A major difference between Flexible BFT and snap-and-chat protocols is that Flexible BFT only guarantees consistency when clients have the same beliefs, whereas snap-and-chat maintains a common prefix regardless of client beliefs.

Snap-and-chat protocols use an off-the-shelf dynamically available protocol, Π_{LC} (LC for longest chain), and an off-the-shelf partially synchronous BFT protocol, Π_{BFT}, and run the two subprotocols in parallel. If the underlying BFT protocol provides accountability, then snap-and-chat can provide accountable safety as well [374]. Transactions are input to Π_{LC}, which outputs a growing ledger LOG_{LC}. Nodes periodically take snapshots of LOG_{LC}, which are then input to Π_{BFT}, which spits out a growing ledger of these snapshots, LOG_{BFT}, in an attempt to finalize some of the transactions. The finalized ledger, LOG_{fin}, is formed by concatenating the snapshots in LOG_{BFT} and then removing duplicate transactions. To create the dynamically available ledger, LOG_{da}, the finalized LOG_{fin} is prepended to LOG_{LC} and sanitized again. This sanitization breaks typical light clients, but a more complicated light client design is presented in [378].

Clients who want availability follow the LOG_{da} ledger, not LOG_{LC}. These two ledgers are equivalent under favorable network conditions, but under less favorable conditions, LOG_{fin} and LOG_{LC} may diverge. In this case, by prepending LOG_{fin} to LOG_{LC}, the finalized ledger is guaranteed to be a prefix of LOG_{da} and remain safe so long as Π_{BFT} remains safe. This ensures that all clients eventually agree on a single history regardless of the chain they follow. A malicious party could attempt to break safety by using a fake snapshot with unconfirmed transactions as input to Π_{BFT}. To prevent this, Π_{BFT} requires a slight adjustment, where honest nodes refuse to accept the finalization of snapshots that the node does not believe are confirmed in their view of Π_{LC}.

Around the same time that the snap-and-chat construction was proposed, [379] described a protocol with similar goals called the checkpointed longest chain. Like snap-and-chat, the checkpointed longest chain allows clients the flexibility to choose whether they prefer guaranteed finality or availability. Unlike snap-and-chat, however, the checkpointed longest chain construction provides *coupled validity*, wherein finalized blocks exist on a single linear chain, so one can verify a transaction based on the information from the blockchain leading up to the block that the transaction is included in. Because of this, block proposers know which transactions are valid when they propose blocks, which assures them that the transaction fees included in the block will be received. It also enables simple light clients who merely verify that a transaction is included in a block while making the network more efficient by not processing invalid or duplicate transactions. In [379], four desirable properties of a checkpointing protocol are described, where the properties hold under network synchrony:

1. **Safety**: Even during periods of asynchrony, honest participants checkpoint the same block in an iteration of the protocol, and this checkpoint lies on the same chain as all prior checkpoints.
2. **Recency condition**: A newly checkpointed block must have been at an exact depth k in the chain of an honest user within the recent past. If arbitrarily old checkpoints could be issued, many honestly mined blocks could be overwritten, so this property bounds the number of honest blocks that may be reorged.
3. **Gap in checkpoints**: There must be a large enough interval between checkpoints that the recency condition holds. This limits the frequency with which honest blocks may be overwritten by checkpoints.
4. **Conditional liveness**: If all honest nodes share a common prefix in all but a few blocks, then a new checkpoint will be issued within a bounded time.

The checkpointing subprotocol used in the checkpointed longest chain protocol is just a slight modification of Algorand's BA$\star$ protocol (described in Section 13.4.2) extended from single-shot BA to a repeated version. BA$\star$ satisfies all four properties, but many alternative BFT algorithms do not. For example, PBFT and HotStuff fail the recency condition, which provides an adversary with a powerful attack: lock on a block privately while the leader of a view, but wait to finalize it until it is not in the longest chain anymore (despite being a descendent of the most recent checkpoint). This attack would cause many honest blocks to be invalidated.

Of the above listed properties, Afgjort only requires safety, though it adds two other desired properties. Afgjort requires that the finalized chains held by honest nodes keep up with the underlying blockchain's chain growth, which is the opposite of having a sufficiently

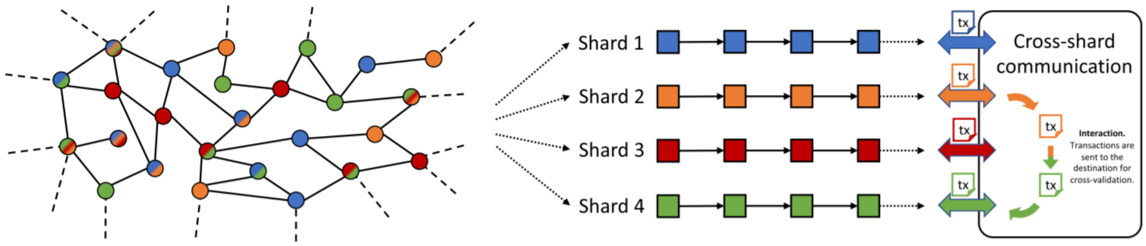


Fig. 46. Sharding architecture. The system is partitioned into groups that each maintain separate shards with their own distinct ledger and transactions. A cross-shard transaction protocol is needed in order for shards to interact safely in parallel. In some sharded systems, a single node may participate in multiple shards, displayed here as multicolored circles. [396]

large gap in checkpoints, as required by the checkpointed longest chain. Both protocols provide $\frac{n}{3}$-support. The properties of GRANDPA include safety, the recency condition, and conditional liveness, but it does not satisfy the gap in checkpoints property. GRANDPA tries to finalize blocks close to the tip of the chain, where blocks may not yet be in a common prefix of honest nodes. Because honest nodes might have different views of blocks near the tip, it is likely that more honest blocks must be discarded when they are found to conflict with blocks that are ultimately finalized.

15. Sharding

Sharding is a scaling methodology from traditional database systems. In the context of state machine replication, the goal of sharding is to divide the required network communication, state data storage, and transaction processing computations across different subsets of validating nodes. In this way, a given node need not maintain or process the entire log of the state machine. There are many academic and commercial sharding proposals, such as Elastico [380], OmniLedger [381], RapidChain [382], Monoxide [383], Zilliqa [384], NEAR Protocol [385], Ethereum 2.0, and more [386–393]. The basic idea of sharding is shown in Figure 46. A sharding protocol that implements state machine replication typically contains the following subcomponents [394, 395]:

- Identity establishment and committee selection, which may include registering public keys or IP addresses and submitting proof-of-work solutions or other Sybil-resistance measures
- An overlay setup for committees so that committee members may communicate with one another
- Intra-committee consensus to agree on a set of transactions within a shard
- Cross-shard transaction processing to maintain atomicity when a transaction requires reading or writing the state of multiple shards

- Epoch reconfiguration, which generally requires unpredictable and unbiasable randomness

Essentially, a sharding protocol will take a registered set of validators and securely sort them into committees responsible for validating particular shards. Each committee executes a consensus algorithm to agree on a transaction log within a particular shard. Transactions that take place across shards must be handled with extra care. Periodically, the committees need to be re-shuffled in order to prevent an adversary from taking over individual shards. Depending on the scheme, there may be some fixed number of shards, or more shards may be added as new validators join the network. Designing a scalable and secure sharding protocol is non-trivial, and each one of the above components has its own challenges and a wide design space.

There is a limit to how much sharding can improve the scalability of state machine replication. In particular, sharding cannot improve scalability in the face of a fully adaptive adversary but can scale from $O(n)$ to $O(\frac{n}{log\ n})$ against mildly adaptive adversaries (i.e., corruptions are determined at the beginning of each epoch and static throughout), where n is the number of validators in all shards [397]. This improvement requires the system to provide a mechanism to create succinct proofs that the epoch's state updates were valid (e.g., issuing checkpoints or some form of verifiable computation) at the end of every epoch. Scalability is greatly impacted by how many transactions take place across shards and how many shards they involve. If validators are required to keep track of the information in the corresponding shards of all cross-shard transactions, then the sharding protocol does not (asymptotically) improve scalability [397].

15.1. Intra-Shard Consensus

The participants are split into shards that each independently run their own consensus algorithm. One of the primary challenges in sharding design is to ensure that validators are partitioned into committees such that all committees satisfy the security threshold of the underlying consensus mechanism with overwhelming probability. Access to unpredictable randomness is critical to this process and is discussed in more detail in Section 15.2. Because there are fewer validators per shard, it is more challenging to prevent individual shards from being taken over by an adversary than it is to secure an unsharded network.

Most sharding proposals use a permissioned BFT algorithm to maintain consensus within an individual shard. One of the first sharding systems proposed, Elastico, simply uses PBFT. Unfortunately, because PBFT's communication complexity is $O(n^4)$ in the worst case, shards are unable to support many validators. This leads to an unacceptably high failure probability for individual shards at the claimed security level, which tolerates up to $\frac{1}{4}$ of the computational power of the network being Byzantine (where proof of work is used to register identities). Elastico was tested with shards of only 100 validators. Based on the cumulative binomial distribution, the probability of having at least 34 Byzantine validators assigned to a shard is 2.76% per shard.

OmniLedger improves upon this by suggesting a more scalable BFT protocol called ByzCoinX. In ByzCoinX, the communication pattern is changed so that instead of every validator being pairwise connected, validators are assigned evenly to groups within each shard. The protocol leader assigns one validator in each group to be a group leader in charge of communicating with the protocol leader on behalf of the group members, and new group leaders are chosen if they do not respond quickly enough. Within each group, signatures are aggregated to reduce communication complexity, and once the protocol leader gets responses from $\frac{2}{3}$ of group leaders, they can move on to the next phase of the consensus protocol. This pattern allows far more validators to participate in a shard and is, thus, able to attain security when up to $\frac{1}{4}$ of the computational power is Byzantine.

RapidChain is able to improve this threshold to tolerate up to $\frac{1}{3}$ of the computational power being adversarial by using a synchronous BFT protocol (secure under an honest majority of participants) for intra-shard consensus. While this does allow a shard to remain secure with fewer validators, cross-shard transactions can cause consensus failure if the synchrony assumption is violated.

The division into small committees presents other security issues. The incentives for validators to behave properly within a shard are understudied, and for a simple scheme where rewards are equally split among participating validators, the Nash equilibrium includes not participating in committee tasks like validation and message passing [398]. Worse, if the adversary is capable of bribing committee members, it can be substantially cheaper to bribe and corrupt enough validators on a particular shard than to take over the entire system. If an invalid state transition occurs in one shard, invalid state may propagate to other shards via cross-shard transactions, which can allow actions like spending funds that do not exist. Fraud proofs, discussed in Section 15.5, are an attempt to remedy this.

15.2. Identity Registration, Committee (Re)configuration, and Epoch Randomness

Like the other systems described in this document, some kind of Sybil-resistance mechanism is required to participate in a sharded ledger system. Many systems employ a global ledger (sometimes called the beacon chain, identity chain, reference chain, etc.) to keep track of these identities and other shard-related metadata.

In Elastico, the number of committees grows linearly with the total computational power deployed on the network. In the first step of every epoch, validators generate an identity by completing a proof of work that covers a public key, IP address, and the epoch's random seed. An identity is assigned to a committee of c validators based on the least significant bits of the proof-of-work solution. Upon creating an identity, a validator must figure out which other validators are assigned to the same shard in order to establish point-to-point connections with them. To reduce the bandwidth of communicating these identities, parties contact a *directory committee* composed of the first c identities created within the same epoch according to that party's view and receive at least $\frac{2c}{3}$ lists of c identities. The validator then considers the union of these lists to be their committee. This leads to discrepancies in validators' knowledge of who else is in the same committee, but these discrepancies are bounded. Each shard runs its intra-committee consensus algorithm on disjoint transaction sets and then sends the valid transactions to the *final committee*. The final committee merges the transactions from each shard, which is then broadcast to the network.

The last step of the epoch requires the final committee to generate and broadcast the next epoch's random seed using a commit-and-XOR protocol. Each member of the final committee generates an r-bit random string, R_i, and sends the hash $H(R_i)$ to the other members. The committee executes a consensus instance to agree on a set S of at least $\frac{2c}{3}$ hashes, and then S is broadcast to the network. Next, each final committee member reveals their R_i preimage to the complete network. At this point, every node will have received between $\frac{2c}{3}$ and $\frac{3c}{2}$ random strings, discarding ones that do not match their hash. In the next epoch, users determine their random seed by XORing any $\frac{c}{2}+1$ of the R_i. Since nodes may choose different R_i, nodes must include the set of random strings in their identity so that others may verify that they match the commitments in S.

There are a number of weaknesses to this scheme, which are improved upon in later designs. Firstly, the commit-and-XOR randomness generation process can be biased by attackers. In addition, validators will frequently switch shards, which reduces scalability; switching only some nodes per epoch would help. This is compounded by the need to store and propagate every transaction in the system from every shard. Only the transaction validation and execution process is improved, but neither storage nor bandwidth improve from sharding this way. As a result, there are no cross-shard transactions in Elastico. Finally, Elastico requires a trusted setup to generate the initial epoch randomness, which must be revealed to all parties at the same time.

OmniLedger improves upon Elastico in a number of ways. Identities are generated similarly and then committed to a global *identity blockchain*, where the commitment must occur the epoch prior to a validator participating in the system. In each epoch, a new, bias-resistant random seed is generated using a VRF in a manner similar to that used in Algorand (Section 13.4.2). The VRF output is used to elect a leader to run a distributed random beacon protocol called RandHound, which is then used to create the epoch's random seed.

More specifically, let e be the epoch, $config_e$ be the list of registered validators from the identity blockchain, and v be the view. Then, when epoch e begins, each validator i computes $ticket_{i,e,v} = VRF_{sk_i}(''leader''||config_e||v)$ and broadcasts it to the network. After waiting for Δ time, validators accept the lowest valued ticket as the leader for RandHound. Validators wait another Δ time for the leader to initiate RandHound, incrementing the view number if the leader fails. Eventually, RandHound will output a random seed and correctness proof, which is broadcast to all registered validators. The random value is used to permute the registered validator list, which is then divided into roughly equal-sized committees. To maintain liveness during epoch changes, shards are reconfigured only in small batches of at most $\frac{1}{3}$ of the shard size at a time.

Crucial to OmniLedger's performance is the idea of *state blocks*, which commit to the complete state of the shard and are analogous to checkpoints in classic BFT systems like PBFT. At the end of each epoch, the state block is appended to the shard's chain and points to the previous epoch's state block. When a validator switches shards, they do not execute every transaction included but rather bootstrap their state based on the state block. This enables transactions from older epochs to be pruned from the shard chain.

OmniLedger still requires a trusted setup to generate the initial random seed for the VRF. RapidChain solves this problem with a secure bootstrapping subprotocol that uses verifiable secret sharing (VSS). During bootstrapping, the initial set of RapidChain participants agree on a set of $O(\sqrt{n})$ nodes to be the *root group*. The root group generates a random seed using VSS to be used to elect a *reference committee*, C_R, of size $O(log\ n)$. The reference committee then creates k more shards of similar size by hashing participants' identities to the range [0,1) and partitioning this space into k regions. To create the initial root group, RapidChain participants are divided uniformly at random into groups of size $O(\sqrt{n})$ using a deterministic process. Each group creates its own random seed using VSS. Within each group, all participants hash the seed and their public key. A small constant number of the lowest hashes from each group are elected and make up the root group. To inform everyone else, these hashes are gossiped to other groups with at least half of the signatures of the group.

Identity registration in RapidChain is performed using proof of work that incorporates the prior epoch's randomness. Upon solving a proof-of-work puzzle, a node submits it to the reference committee to be included in a *reference block*, which contains the active identities for the next epoch, the shards they are mapped to, and the next epoch's randomness. The reference block is then sent to each committee.

The epoch reconfiguration protocol in RapidChain is based on the *Cuckoo rule*, which is safe because it only allows a constant number of validators to switch their committee per epoch. This makes RapidChain resistant to attacks where the adversary strategically attempts to join or leave the network in order to concentrate their power within a target shard. As a result, committees can be reconfigured more frequently than with OmniLedger. The Cuckoo rule works as follows: new identities that join the network are mapped to a random position $r \in [0,1)$. For some constant x, identities within the interval $(r-x, r+x)$ are moved to new random positions. Partitioning nodes into k groups of $O(log\ n)$ has been proven secure against an adversary with $\leq \frac{1}{2} - \frac{1}{k}$ of the computational power. That is, each committee will have a bounded number of Byzantine nodes.

The issue of shard (re)allocation was studied more formally in [399], where the Wormhole protocol was proposed. This work describes two performance metrics for sharding allocation:

1. *Self-balance*: Ideally, nodes would be uniformly distributed among shards. Other-

wise, shards with fewer nodes will have weaker fault tolerance, while shards with more nodes will have worse performance. While it may not be possible to attain perfect load-balancing in a permissionless network due to nodes freely joining and leaving, the optimal load balance can be achieved by having a random subset of nodes move to other shards. Self-balance, then, is the ability of the sharding protocol to recover when the load becomes imbalanced.

2. *Operability*: When nodes move from one shard to another, they must synchronize their state to match that of the blockchain in the new shard. This process can have high overhead and reduce these nodes' availability while synchronizing. Operability measures the cost of performing this relocation to another shard.

Unfortunately, it is impossible for a shard allocation protocol to simultaneously satisfy both optimal self-balance and optimal operability. Existing protocols tend to fall into extremes on either self-balance or operability. For example, Elastico and OmniLedger optimize self-balance at the expense of operability, while Monoxide (Section 15.4) has optimal operability but lacks self-balance. RapidChain does not fall into either extreme but has inferior performance compared to Wormhole. Luckily, there is another property, *non-memorylessness*, which allows a shard allocation protocol to parameterize between the two metrics. A protocol is non-memoryless if each shard allocation does not rely solely on current and incoming system states but also takes into account prior system states. Wormhole has this property and, thus, allows a system designer to parameterize this trade-off instead of picking one to optimize.

Wormhole (Figure 47) assumes the existence of an external randomness beacon and uses the beacon output as input to a VRF. The VRF output determines shard allocation. Each randomness beacon output updates the system state used for non-memorylessness. However, the more prior system states that are used, the more information it takes to prove membership in a shard. Furthermore, the shard membership proof in epoch r depends on the proof in epoch $r-1$ in a recursive manner. Wormhole prevents this proof from becoming unbounded in size by using a parameter, w, that controls how often old proofs are discarded. An *era* consists of w epochs, where each epoch corresponds to a new beacon output. Each node must have one non-memory-dependent allocation epoch per era in order to discard old proofs. If all nodes were to discard their historical proofs and have this non-memory-dependent allocation in the same epoch, operability would be significantly impacted as many nodes would change their shards simultaneously. Instead, Wormhole randomly assigns nodes to non-memory-dependent allocation epochs. When w is larger, nodes joining the system will need to execute the VRF more times, making it more expensive for an adversary to join a target shard by trying to join the system repeatedly.

Wormhole uses an operability parameter, op, to manage the trade-off between operability and self-balance. When op is large, there is only a low probability that a node will be moved to another shard. To determine which shard a node is allocated to in memory-dependent epoch r, nodes consider the most recent VRF outputs since their latest non-

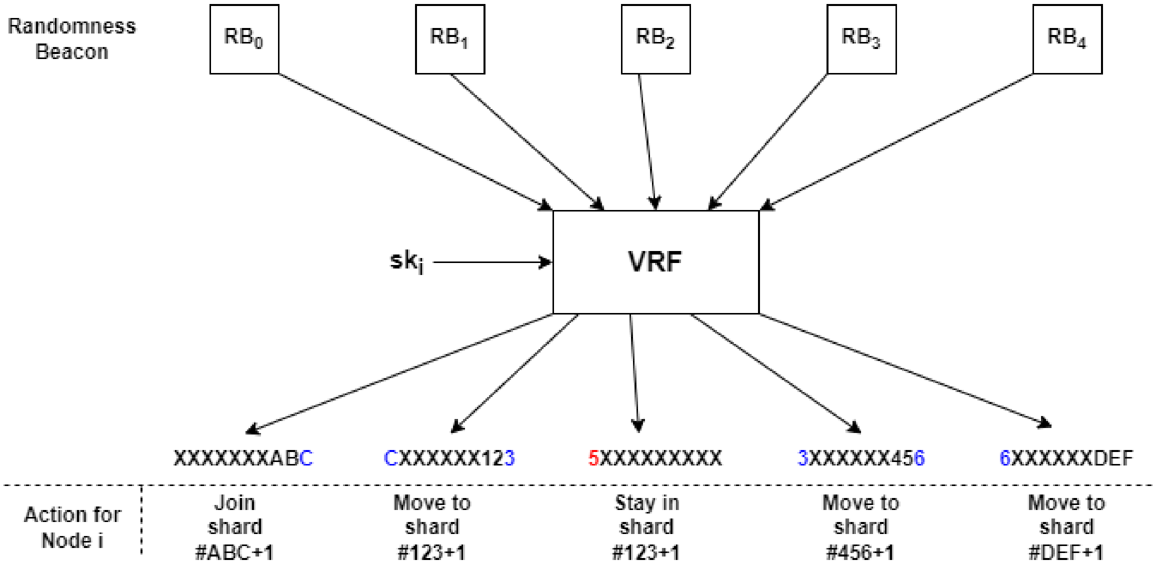


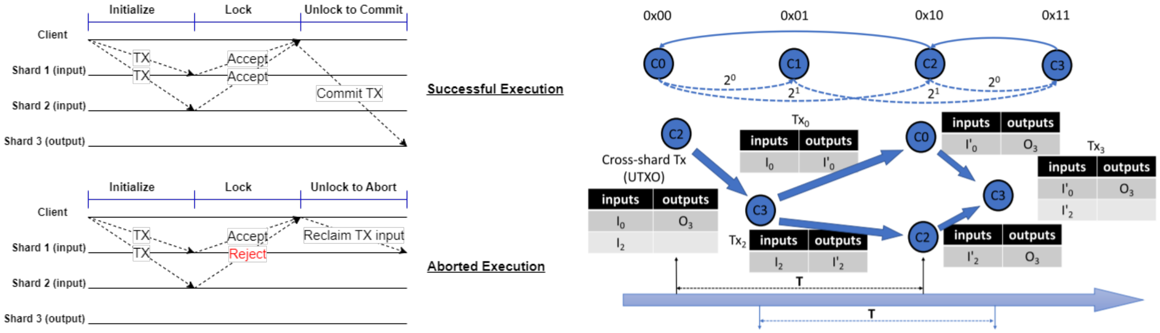
Fig. 47. Wormhole shard allocation. Simplified example in hexadecimal with parameters $op = 4$, $w = 5$, and $m = 16^3$ shards. Epoch 0 is the most recent non-memory-dependent epoch for node i, so they join a shard based solely on the VRF output. In the other $w - 1$ epochs, node i compares the op most significant bits of the new VRF output with the op least significant bits of the output that most recently assigned them to a new shard. Matches are shown in blue, and misses are in red.

memory-dependent epoch, x. Let m be the number of shards and $\{h_x, ..., h_r\}$ be the set of VRF outputs in memory. Then, for $j \in [x+1, r]$, nodes check whether the op most significant bits of h_j match the op least significant bits in the VRF output from the most recent epoch where a node changed shards. If so, the node switches to shard $(h_r \, mod \, m) + 1$, and if not, the node remains on the same shard.

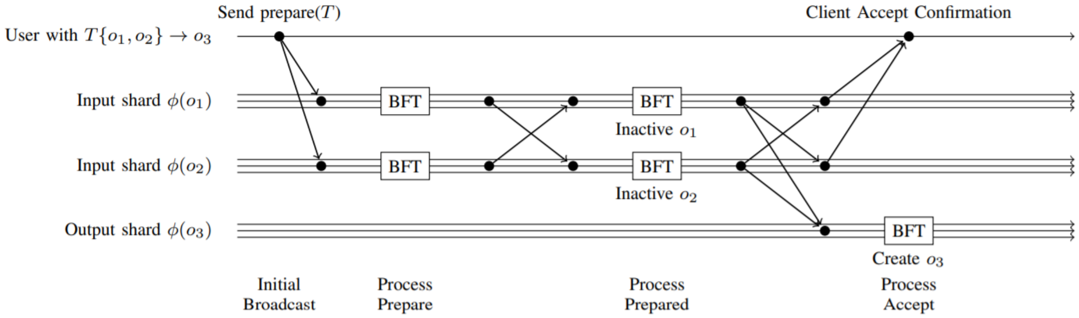
15.3. Cross-Shard Transaction Processing

In order for sharding to improve a system's throughput, it is necessary for transactions to be partitioned across shards such that not all nodes need to process every transaction or maintain the full system state. As a result, many transactions are likely to take place across multiple shards: if Alice's account is on shard A and Bob's account is on shard B, then Alice and Bob will not be able to transact without impacting the state of both shards. For this to be safe, sharding systems require an *atomic commit* for the state changes that occur across shards; either both accounts are updated, or neither is. The most common algorithm for solving this problem is the two-phase commit (2PC) algorithm, and the protocols in this section follow that paradigm.

OmniLedger presents a UTXO-based cross-shard transaction subprotocol called Atomix. In order to remove the need for direct shard-to-shard communication, Atomix tasks the



(a) Atomix **(b)** RapidChain Cross-Shard Transactions



(c) S-BAC

Fig. 48. Cross-shard transactions. (a) Atomix. (b) RapidChain. The top panel shows each shard maintaining a routing table with $O(log\ n)$ other shards to improve communication efficiency. Committee C_0 can reach C_3 via C_2 for transactions beginning with 0x11. The bottom shows a cross-shard transaction being split into three pieces. (c) S-BAC. Transaction T has two inputs, o_1 and o_2, and one output o_3. The client sends T to all nodes in the input shards. A designated leader in each shard sends either a $prepared(accept, T)$ or $prepared(abort, T)$ message to the nodes within their shard. The leader of each shard determines whether all shards are in a state of $proposed(accept, T)$ or if there are any in $proposed(abort, T)$, handles the $accept(T, *)$ messages, and sends the decision to the client. [386, 396]

transacting client with the responsibility of driving the process forward but allows other parties to help if a transaction is stalled. Atomix is a three-step process of initialization, locking, and unlocking, shown in Figure 48a. Atomix is initialized by having the client create a cross-shard transaction and gossiping it to each shard responsible for the transaction inputs. If the transaction is valid, it is included in a shard block, and a Merkle proof of inclusion is the transaction's proof of acceptance. If the transaction is invalid, a proof of rejection is created instead by setting a bit in the block. The client gathers proofs of acceptance from each input shard and can communicate them to the output shards who can then generate the needed UTXOs. Alternatively, with a proof of rejection, the client can abort the transaction to unlock their funds on the input shards.

RapidChain adopts a different approach to cross-shard transactions in the UTXO model.

To reduce the amount of communication used between shards, a Kademlia-like overlay network is used to route cross-shard transactions to the appropriate shards, as shown in Figure 48b. Cross-shard transactions in RapidChain are broken into multiple transactions. For instance, for a two-input one-output transaction with inputs I_1 and I_2 from shards one and two and output O in shard three, the transaction in Figure 48b will be executed as three different sub-transactions:

- tx_1 consumes input I_1 and creates output I'_1 belonging to shard three.
- tx_2 consumes input I_2 and creates output I'_2 belonging to shard three.
- tx_3 consumes inputs I'_1 and I'_2 and creates output O.

In effect, tx_1 and tx_2 transfer I_1 and I_2 to the output shard, which are then spent in tx_3 to create the intended output O. Each sub-transaction occurs on a single shard, and the committee for the output shard will route the input transactions to their relevant shard committees. If, say, tx_2 were to fail while tx_1 was successfully committed, the owner of UTXO I_1 instead uses I'_1 in a future transaction.

Another approach, S-BAC (Sharded Byzantine Atomic Commit), was proposed for the sharded smart contract platform Chainspace [386]. S-BAC is similar to Atomix but eschews the client-driven model in favor of one that explicitly runs a Byzantine agreement algorithm combined with atomic commit, as shown in Figure 48c. The client sends their transaction, T, to the input shards, which internally execute PBFT in order to agree on a tentative decision to accept or abort the transaction. Those tentative decisions are then broadcast to the other shards involved in the transaction as $prepared(abort, T)$ or $prepared(accept, T)$ messages that include signatures from shard members proving they decided a particular way. If the transaction was locally accepted, the input is considered locked. Shard members listen for responses from the other shards involved. If all responses support accepting transaction T, then it is committed, but if any shards want to abort, the transaction is aborted. The shards then exchange a round of $accept(commit, T)$ or $accept(abort, T)$ messages based on their decision and send them to the client as well. Once the transaction is committed, the output shards generate new outputs, and the inputs are consumed. If the transaction is aborted, input shards unlock the inputs.

Both S-BAC and Atomix were susceptible to replay attacks that do not require violating Byzantine thresholds [400]. It is crucial for a system design for cross-shard transactions to defend itself against replay attacks in order to prevent invalid state from being incorporated into the ledger. The attacks had two causes, which informed their fixes:

1. The input shards do not have a way of knowing that particular protocol messages received correspond to a specific instance of a transaction, so old messages can be replayed. To fix this, sequence numbers are added to transactions.
2. In some cases, the output shards are only involved in the later unlocking phase of the protocol and therefore have no knowledge of the transaction context that is available

to the input shards. To fix this, output shards create dummy objects in the earlier locking phase, which makes them input shards as well.

A concern related to cross-shard transactions is how to partition the system state across shards. Because cross-shard transactions have higher latency and require more effort from the network, dividing the state in such a way that accounts that frequently interact with each other are located on the same shard would improve the system's performance. On the other hand, partitioning the state this way can lead to very memory-inefficient mappings between shards and the state they are responsible for. Most schemes partition the state based on a simple mapping of account prefixes to shards, which is essentially random. Assuming that there are 100 shards, one would expect 99% of transactions to be cross-shard. With only 10 shards, this still results in 90% of transactions being cross-shard. A few alternative mapping strategies that can reduce the fraction of cross-shard transactions while being relatively memory-efficient are explored in [401]. This includes ideas such as using graph clustering on accounts that historically have interacted with each other (though this is an NP-hard optimization problem), clustering the most heavily used accounts and assigning the rest randomly, and clustering based on the most frequently used accounts during a recent time period.

15.4. A Different Approach: Monoxide

Monoxide proposes a very different approach to sharding from what has been discussed so far [383]. Unlike the above examples, identities in Monoxide are established only once, and there is no committee reconfiguration. Monoxide uses accounts rather than UTXOs, and accounts are partitioned based on their most significant bits. Each individual shard uses proof of work and the GHOST fork-choice rule (Section 11.2) for consensus. Every node in Monoxide must be a light client of every shard and maintain a chain of block headers for each. Nodes also maintain a distributed hash table (DHT) for routing cross-chain transactions and for peer discovery.

The most noteworthy aspect of Monoxide is its use of *chu-ko-nu mining*, which is similar to the concept of merged mining (see Section 16.2). Chu-ko-nu mining allows miners to create blocks in multiple shards simultaneously with a single proof of work, allowing honest miners to amplify their hash rate and prevent malicious miners from concentrating their computational power into taking over single shards. If every honest miner takes advantage of this and mines on every shard simultaneously, then Monoxide attains security assuming an honest majority of the hash rate. Miners will gather valid transactions from all shards and include a Merkle root of the block headers for each shard in their proof of work.

While this mechanism increases the adversarial threshold required to attack Monoxide compared to other sharding protocols, it is imperfect. A miner needs to verify all transactions in order to participate in all of the shards and achieve the honest majority security bound, but this eliminates the scalability benefits for those miners. The result is likely to be severe centralization pressure among miners, since well-resourced miners can mine on more shards and collect more transaction fees.

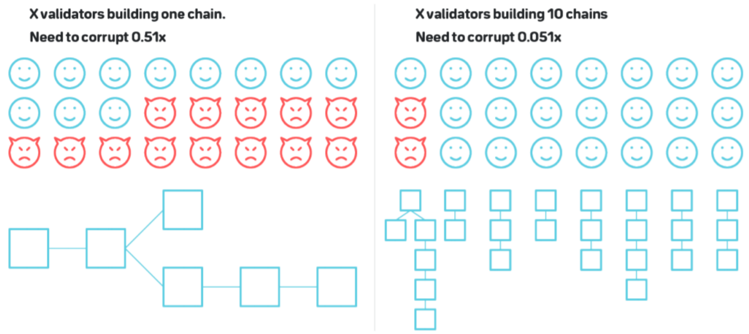
Monoxide also takes a different approach to cross-shard transactions, bypassing the overhead of the locking and unlocking operations used in other protocols. Instead, Monoxide accepts that transaction atomicity only holds eventually with high probability. As a result, a credit can happen on one shard before the corresponding debit on a different shard has been fully settled. Transactions are validated in the shard of the payer and then verified in the shard of the payee using the header chain of the payer's shard and and a *relay transaction*. The relay transaction exists on the payer's shard and provides any metadata required to validate the original transaction using only the shard's header chain. A miner in the payee shard then verifies that the relay transaction is stable and includes it in a block on that shard as well.

While interesting, Monoxide has some challenges. In addition to the mining centralization pressure, it may be challenging to handle transaction fees, particularly for cross-shard transactions. Similarly, keeping track of inflation of the money supply is difficult or impossible without completely validating all transactions on all shards. Finally, the requirement to act as a light client for all shards makes it so that Monoxide does not scale asymptotically, though it may offer a significant constant factor improvement.

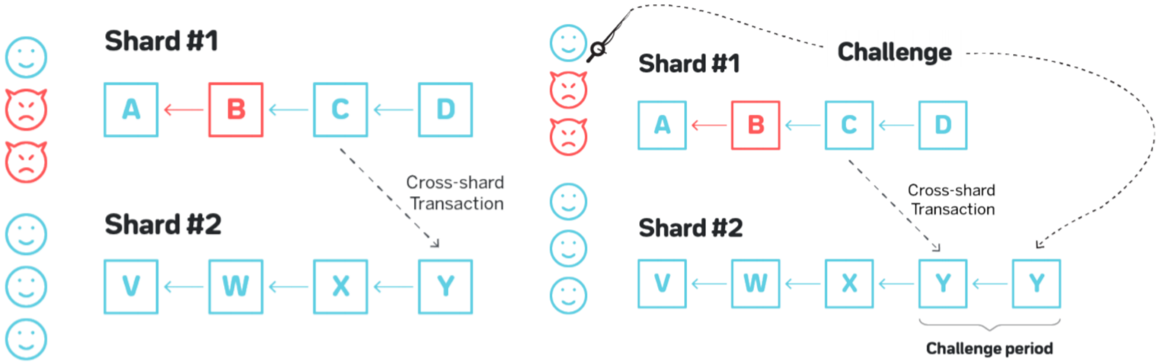
15.5. Fraud Proofs and Data Availability

In a sharded blockchain system, the first challenge to resolve is the so-called *1% attack*, where an adversary with a small fraction of the total network resources (e.g., work, stake, validated identities, etc.) can concentrate them in a single shard and exceed its Byzantine threshold, as seen in Figure 49a. These types of attacks are typically prevented using a form of random sampling to assign validators to shards in an unpredictable way, such as the mechanisms described in Section 15.2. Unfortunately, reshuffling has high overhead. Each time a validator is assigned to a new shard, it must download the state of that shard in order to validate a new block, which can take an extended period of time. This prevents reshuffling from happening too frequently. Naively, this would require downloading and executing all shard blocks that have been produced since the last time the node was assigned to that shard, in which case sharding has provided no scalability benefit. One step toward mitigating this is to have every shard block's header commit to the state of the shard, so nodes need not store the complete system state at all times. However, it does not ensure that the state at the tip of the chain is correct, which would require processing all of the shard blocks anyway.

On the other hand, the frequency with which shard reallocation is performed relates to how adaptive of an adversary the system can tolerate. When reallocations are infrequent, the adversary is given a lot of time to (adaptively) target and corrupt validators on individual shards, reintroducing the risk of single-shard takeover attacks. In most of the systems described above, the shards implicitly trust each other. That is, nodes in shard A simply assume that no invalid state is ever committed to shard B. This may not be realistic in the



(a) Single shard takeover attack



(b) Invalid state transition

(c) Fraud proof

Fig. 49. Invalid shard state transition after a single shard takeover attack. An invalid state transition occurred in block B, perhaps crediting an account with undeserved tokens. A cross-shard transaction moves these counterfeit coins to another shard. Any honest node on the corrupted shard can generate a fraud proof and submit it to the victim shard to demonstrate that the state transition was invalid and should be rolled back. [385]

real world, where adversaries may indeed be adaptive and can even bribe validators. In this case, the security of the protocol decreases linearly in the number of shards. When invalid state is committed to a shard, it can then influence the rest of the system via cross shard transactions, as shown in Figure 49b.

To provide security against fully adaptive or bribing attackers, a sharding system should provide validators with the ability to quickly see if a proposed shard block is valid or not. Security in this model is possible using *fraud proofs*, where a node supplies proof of an invalid state change, and adding an *any-trust assumption* on the validators that were assigned to the shard in the last round. That is, as long as there is a single honest and available validator assigned to the shard, it can construct a fraud proof in response to an invalid block being created on their shard, as shown in Figure 49c. A fraud proof may contain the invalid transaction and some Merkle tree data verifiable from the shard block header's state commitment. In other words, a fraud proof will contain the relevant portions

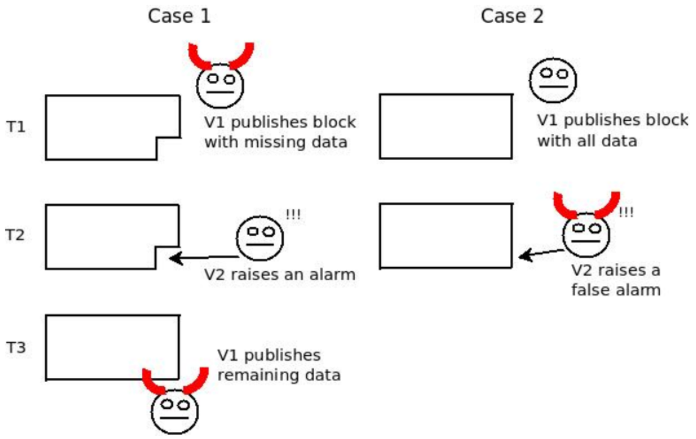


Fig. 50. Data availability attack. The left and right sides are indistinguishable after T_3. [402]

of the state needed to process the block, as well as intermediate hashes of the Merkle tree that prove that the provided state is in fact the state that the block claims to be using [402]. For example, for an invalid transaction that debits 100 units of currency from an account possessing only 50 units, the fraud proof would include the transaction itself, an assertion that the account's balance before the transaction was only 50 units, and a Merkle proof showing that this account balance was committed in the state root of the block header.

For this to work, there must be some bounded challenge period where an honest validator has sufficient time to download and process the block, prepare a fraud proof if it is invalid, and send it across the network. While the challenge period must be long enough to allow invalid shard blocks to be caught, longer periods delay the settlement of cross-shard transactions. Furthermore, this creates a new attack vector where malicious nodes send invalid fraud proofs. This can be mitigated by requiring stake deposits from validators; if a node sends an invalid fraud proof, they lose their deposit. Similarly, if a node submits a valid fraud proof, the creator of the invalid block loses their deposit, and a portion of it can go to the node who created the proof as a finder's fee.

One more major problem remains. Consider an adversary who controls a supermajority of the validators on a shard. This adversary creates an invalid block and signs off on its validity through the validator keys they control. Now, the adversary announces the block by broadcasting its header but withholds some or all of the block data itself. In this case, the honest validators on the shard are unable to produce a fraud proof and can only claim that block data is being withheld. Unfortunately, after an honest node makes this claim, the adversary can immediately publish the data, and other nodes will be unable to distinguish between a situation where the block producer maliciously withheld the block or a malicious validator raised a false alarm, as shown in Figure 50.

This *speaker-listener fault equivalence* prevents the system from punishing false data availability alarms or rewarding valid ones. As a result, under adversarial conditions, all validators would have to download all shard blocks, eliminating any scalability gains. To see this, suppose an attacker executes this attack (Case 1 in Figure 50). If the expected return from raising an alarm were positive, this would encourage malicious validators to submit false alarms frequently and pocket the proceeds. If the expected reward were neutral, then this false alarm method provides a free denial of service to force everyone to download all shard blocks. If the expected reward were negative, then raising the alarm would be altruistic and irrational. An adversary could simply outlast the altruists and then launch data availability attacks while honest nodes have no recourse [402].

This attack is why *data availability proofs* are required for effective sharding protocols in models that assume fully adaptive or bribing attackers. These proofs can be generated using erasure codes, where a block M chunks in size is expanded into N chunks, $N > M$, such that any M chunks are sufficient to recover the original data. Block headers include a Merkle root committing to these N chunks. Light clients can then contact fully validating nodes to check whether the majority of the N chunks are available. If the majority is indeed available, this implies one of three possibilities:

1. The full block is available and valid with a correctly constructed erasure code. In this case, the light client should accept the block.
2. The full block is available, and the erasure code is correctly constructed, but the block is invalid. In this case, the light client would expect an honest full node to construct a fraud proof and broadcast it to the network shortly.
3. The full block is available, but the erasure code is not properly constructed. In this case, the light client would expect an honest full node to construct and broadcast a special fraud proof that demonstrates that the erasure code is incorrect.

Essentially, after conducting the data availability check and having it pass, a light client can wait for the duration of some challenge period to see a fraud proof and treat the block as valid if none are forthcoming. A clever attacker may try to beat this by releasing individual chunks of data as clients ask for them. Since light clients will only sample chunks probabilistically, if there are not enough light clients performing this sampling, an attacker can indeed pull this off and trick them into thinking the block is fully available when it is not. This necessitates an additional security assumption: there are enough light clients making data availability queries that it is overwhelmingly likely that the intersection of their requests covers enough erasure-coded data to recover the full block. Erasure coding forces the adversary to withhold a much greater fraction of the block than they otherwise could have to perform malice, while the data availability queries make it harder for the adversary to get away with it.

The adversary still has one more trick up their sleeve: they can honestly answer requests from many light clients but stop responding before the clients can verify enough of the block's availability. In this case, the adversary can still trick some of the light clients into thinking the block is available when it is not. To address this, clients should send their data availability queries through an anonymous network like Tor, so the adversary cannot tell which queries came from the same client.

A relatively efficient proof system using sparse Merkle trees to represent the system state and 2-dimensional Reed-Solomon erasure codes for data availability is presented in [403]. This system encodes k chunks of data as $2k$ erasure-coded chunks and then provides the following guarantees under synchrony with maximum network delay Δ:

- *Soundness*: If an honest light client is convinced that a block is available, then there exists at least one honest full node with the complete block within some delay $k * \Delta$.
- *Agreement*: If an honest light client is convinced that a block is available, then every light client will consider it available within some delay $k * \Delta$.

The fraud proofs from [403] grow logarithmically in size with the size of the block and state, while the availability proofs grow proportional to the square root of the block size. Similar schemes using a new primitive called Coded Merkle Trees have been proposed [404–406].

By combining fraud and data availability proofs, the security of the system can be assured by assuming a sufficient number of light clients querying for data availability, as well as a single honest full node per shard who is capable of producing and broadcasting a fraud proof within the challenge period. If the adversary is capable of linking validator IDs to IP addresses (perhaps using Sybil nodes on the network and performing traffic analysis), they can violate the any-trust assumption by launching denial-of-service attacks against validators who refuse to collude.

Recall that when a validator changes shards, they need to synchronize their state in order to process and verify new shard blocks. If the validator needed to fully execute all of the shard blocks since they were last validating the shard, this could take hours or days. However, given a system of fraud and data availability proofs, and assuming that there are sufficient light clients to verify data availability and the any-trust assumption within the shard holds, this synchronization process can be sped up dramatically: download the new state, verify that it matches the state commitment in the block header, and wait for a single challenge period to see if anyone submits a fraud proof.

Despite the speedup, this still leaves new nodes unavailable for some period of time after being reassigned to a new shard. One mechanism that can mitigate the consequences of this is to abstract different protocol roles to different parties. For instance, the management of data and execution may be split between nodes with different roles: block proposers, data availability notaries, and transaction executors [407]. Block proposers build a chain of shard blocks, and notaries attest to the availability of data from the shard blocks. The block proposers do not perform any state-dependent validation. Executors take the shard chain agreed upon by the block proposers and execute the transactions, skipping over any that are invalid and signing the state. This would allow executors to stay on one shard while block proposers and notaries can be shuffled and reassigned more frequently. Unfortunately, this has several important trade-offs, such as complicating the design of light clients and making block proposers ignorant of the validity of the transactions they include in their blocks, which makes it harder to keep the system design incentive-compatible.

This delay issue can be fully resolved with the idea of *stateless clients* [408]. Roughly, these would be clients that only maintain the state root, not the state itself. Instead of downloading and maintaining the full system state, stateless clients download a block witness composed of Merkle proofs for all of the pieces of state accessed in the block. This allows the node to compute new state roots without needing a full copy of the state. Block witnesses computed this way are quite large themselves, making this scheme impractical. Recent cryptographic advancements, such as Verkle trees, may be able to reduce witness sizes sufficiently to enable stateless clients [409].

16. Interoperability

Due to the proliferation of distributed ledgers that have been and will be deployed, some degree of interoperability is desirable. This can allow for atomically exchanging assets across multiple ledgers, creating "wrapped" representations of assets that can exist on alternative ledgers, and other cross-chain smart contracts [410, 411].

16.1. Cross-Chain Communication, Fair Exchange, and Atomic Swaps

A basic requirement to have interoperability among distributed ledgers is for the relevant ledgers be able to "communicate" with each other. In particular, a ledger should be able to validate that certain events have occurred in another ledger in order to interact with it. More formally, an entity P monitoring ledger L_X and entity Q monitoring ledger L_Y should be able to synchronize with each other such that Q writes TX_Q to L_Y if and only if P has written TX_P to L_X [412]. In other words, the goal of cross-chain communication is to ensure that either both of TX_Q and TX_P are written to their respective ledgers or neither is. An example is shown in Figure 51.

Cross-chain communication is very similar to the atomic commit problem mentioned in the context of cross-shard transactions in Section 15.3, where relevant processes must agree on whether a transaction was committed or aborted. More specifically, *non-blocking atomic commit* (NB-AC) requires that all correct processes decide on a common outcome even when other processes fail. Cross-chain communication is essentially the NB-AC problem, but participants can have Byzantine failures instead of just crash failures.

In fact, it has been proven that cross-chain communication reduces to the *fair exchange problem* [412]. In a fair exchange, two parties want to exchange goods, and either both items are transferred successfully or neither are. In the analog world, this can be thought of as the problem of a customer simultaneously handing cash to the cashier while the cashier hands the customer a purchased product. In the digital realm, this is often associated with

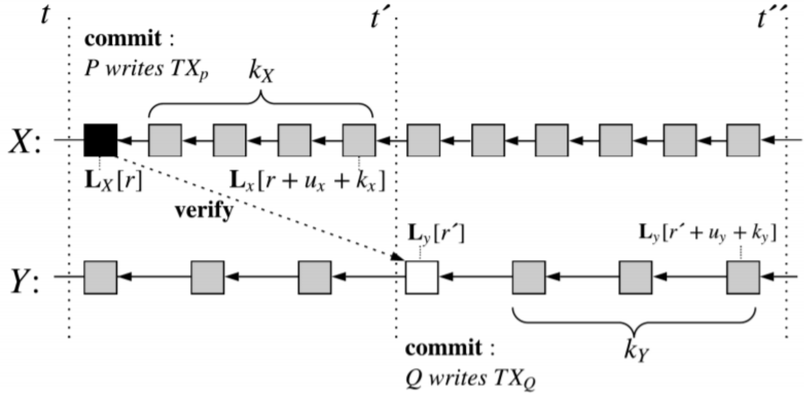


Fig. 51. Cross-chain communication example. Systems X and Y maintain ledgers L_X and L_Y, respectively. A node Q writes TX_Q if and only if P has written TX_P. In this example, transactions are committed in the next block of the ledger once broadcast (liveness delays $u_X = u_Y = 0$). For safety, the persistence delays for systems X and Y are $k_X = 4$ and $k_Y = 3$. [412]

either selling access to files or trading digital signatures. Unfortunately, it has long been known that fair exchange is impossible without a trusted third party in asynchronous networks [413]. Cross-chain communication, therefore, requires either:

- Synchrony between participants such that one ledger can verify the existence of a transaction on another ledger within a known, bounded amount of time or
- A trusted third party, which can be abstracted as a separate replicated state machine.

A line of research has produced protocols for fair exchange that use smart contracts as the trusted third party and often rely on incentives and the rationality of the participants for security [414–418]. These protocols punish malicious users who try to violate the fairness of the exchange such that a rational participant will not do so. However, this does not ensure correct execution of the exchange, even if it prevents honest parties from being harmed economically. While these protocols tend to operate on a single ledger, similar concepts are frequently used in interoperability protocols.

One of the earliest interoperability proposals in the distributed ledger space is the *atomic swap*, where an asset from one blockchain system is traded for an asset on a separate blockchain system without using a trusted intermediary. A specific mechanism for doing so using *hashed timelocked contracts* (HTLCs) was proposed by TierNolan [419], generalized and formalized in [420], and machine verified in [421]. The atomic swap protocol provides the following guarantees:

- When all parties execute the protocol honestly, all swaps will occur.
- If any parties deviate from the protocol, honest parties will not end up worse off.

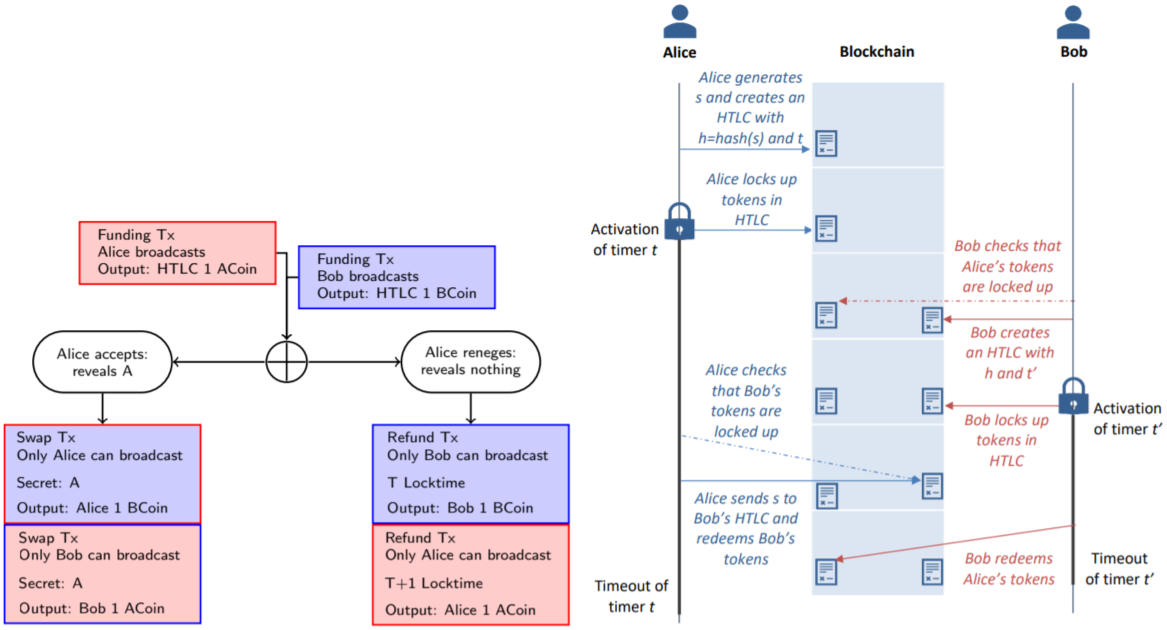
- No party or coalition of parties is incentivized to deviate from the protocol.

An example of a cross-chain atomic swap is shown in Figure 52. Assume that Alice possesses some of an asset, ACoin, that resides on blockchain A, while Bob possesses BCoin on blockchain B. Alice and Bob can perform an atomic swap as follows:

1. Alice generates a random number s and keeps it secret until later.
2. Alice publishes a smart contract with a hash lock $H(s)$ and time lock t_A to blockchain A. If Bob can show s before t_A, the ACoin is transferred to Bob; otherwise, Alice can reclaim it after t_A.
3. Seeing the published value $H(s)$, Bob publishes a smart contract on blockchain B using hash lock $H(s)$ and time lock t_B with $t_B < t_A$. If Alice provides s before t_B, then BCoin is transferred to Alice; otherwise, Bob can reclaim it after t_B.
4. Prior to t_B, Alice will publish s and acquire BCoin from Bob's smart contract. At this point, s is revealed on blockchain B.
5. Prior to t_A, Bob publishes s to acquire ACoin from Alice's smart contract.

Many other atomic swap protocols have been proposed since. For example, JugglingSwap works even when the elliptic curves used for transaction signing differ across blockchains [424]. Another alternative, AC^{3}WN, allows participants to publish their smart contracts simultaneously and thus substantially reduce latency for the swap [425]. Other proposals focus on specific cryptocurrency pairs with unique challenges, such as swapping Bitcoin and Monero [426, 427]. Because Monero does not include a scripting language, different techniques are required. There are also protocols for more general atomic commerce, such as auctions, arbitrage, brokering deals, and escrow-based payments [428, 429]. Atomic debt instruments [430] and options contracts [422] have also been proposed. Finally, there are a variety of cross-chain communication protocols that allow arbitrary smart contracts to function across multiple blockchains [431–436].

A known issue with most atomic swap protocols is that they behave like a free American call option for one of the parties to the swap [437–440]. An American call option is a contract that allows a party to purchase some quantity of an asset at an agreed-upon price at any point prior to the agreed-upon expiry time. The party that initiates the atomic swap is the "buyer" of the American call option but does not need to pay a premium for this option. This can cause problems if the exchange rate between the swapped assets changes during the period that the swap is executed. The option buyer can abort the swap if the change in exchange rate is unfavorable to them. Adding an extra phase where a premium is paid helps mitigate this issue [440].



(a) Swap contracts **(b)** Workflow

Fig. 52. Atomic swap. (a). Each box is a transaction. Boxes with red fill color take place on the ACoin blockchain, whereas those with blue fill color are for the BCoin blockchain. Alice can publish transactions with red borders, while Bob can publish transactions with blue borders. A box directly beneath another box represents a transaction that should be published in response to the one above it. (b). Another representation of the atomic swap workflow. [422, 423]

16.2. Bootstrapping Methods: Merged Mining and Proof of Burn

Merged mining is a specialized form of interoperability where a miner mines on multiple blockchain networks simultaneously. It is primarily a mechanism for bootstrapping a new "child" proof-of-work blockchain by piggybacking on the security of a more mature "parent" chain.

When using merged mining, a valid proof-of-work solution for the parent chain is considered valid for the child, which allows the child chain to inherit the security of the parent chain. Miners construct a block template for the child chain and include a hash of this block in their template for the parent chain. Miners work off of this parent chain template and search for proofs of work at either the (lower) difficulty level of the child chain or the (higher) difficulty of the parent chain. When a valid proof of work at the parent difficulty level is found, it is a normal valid block in that network, and the child chain block hash is ignored by participants in the parent network. If a valid proof of work is found at the lower child difficulty level, a block on the child chain is created and will include the parent chain block header, the coinbase transaction where the child hash is included, and a proof that the coinbase transaction is included in the parent block. This block is submitted to the child chain's network, which accepts the proof of work. Because the parent chain header cannot be created without including the child chain's hash, this still proves that work was performed while constructing the child block. Of course, a block solved at the parent difficulty level will also satisfy the child difficulty level. An example of the merged mining procedure is shown in Figure 53.

There are a number of prominent blockchain networks that employ merged mining, including Namecoin and Rootstock on Bitcoin and Dogecoin on Litecoin [442, 443]. According to [443], some 90% of Bitcoin miners are merged mining as of October 2020, which is a dramatic increase compared to a few years prior. At that time, Rootstock was the most popular chain for merged mining, with 40 to 50% of Bitcoin miners participating.

Merged mining has security implications for both the parent and child chains. Merged mining includes a certain amount of overhead, primarily to perform validation of the child chain. Larger miners on the parent chain are more likely to be able to afford this overhead and thus claim the extra rewards from mining the child chain, which may lead to centralization pressure for the parent chain [444, 445]. An alternative merged mining proposal called *blind merged mining* may be able to address the centralization concerns for the parent chain [446]. This will be discussed more in the next section in the context of the Drivechains protocol [447, 448]. Because validation costs are the primary cost of participating in merged mining, the miners are also less likely to validate the child chain. As a result, the security of the child chain may not be as significant as the hash rate would suggest. In practice, it has frequently been the case that the hash rate on merged mined child chains fluctuates significantly and is highly concentrated [444].

There are additional risks to parent chains, which can present interesting problems given that a parent chain cannot prevent being merged-mined by child chains. This issue is com-

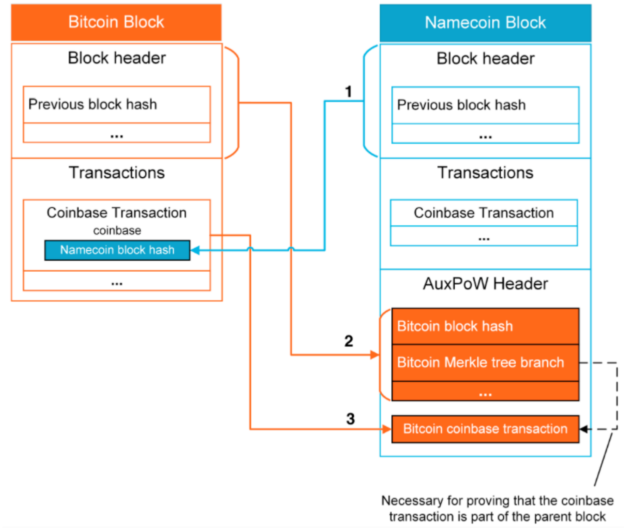


Fig. 53. Merged mining example, where Namecoin is the child chain to Bitcoin. In the first step, a Namecoin block is constructed, and its hash is placed in the coinbase transaction of a Bitcoin block. If a proof of work is found at Bitcoin's difficulty level, the Bitcoin block header, the coinbase transaction, and a Merkle proof of the coinbase transaction are included in a special portion of the Namecoin block. [441]

pounded by the advent of multi-merged-mined cryptocurrencies that have multiple parent chains. A miner for this cryptocurrency could attack a subset of the parent chains at no additional cost [444]. Parent chains can also be attacked when there are (social) disputes over the rules that a ledger should follow, though it would be challenging to pull off such an attack [449].

Another technique that is frequently used for interoperability and to bootstrap new ledgers is *proof of burn*, or the verifiable destruction of a cryptocurrency [450]. To "burn" an asset involves sending it from a source ledger to a provably unusable address. The proof of burn itself consists of the transaction that burned the asset, the block header for the block containing the burn transaction, a (Merkle) proof demonstrating that the burn transaction was included in that block, and a proof that the submitted block is in the "best" source blockchain (e.g., the chain with the most proof of work) and is stable. This technique is used in asset transfer using relay smart contracts, as described in the next section, but also sometimes as a Sybil-resistance mechanism for a separate consensus algorithm.

Related to proof of burn is another bootstrapping mechanism employed in the Stacks blockchain called proof of transfer (PoX), where the security of the Stacks ledger is anchored to Bitcoin [451]. In PoX, instead of burning units of the base cryptocurrency (Bitcoin), validators transfer those coins to owners of the new cryptocurrency (e.g., Stacks) via a predetermined set of Bitcoin addresses as a reward for producing blocks on the new chain. A Stacks block producer commits to their proposed block in a Bitcoin transaction, and one of the various block commit transactions within a single Bitcoin block is chosen via a VRF in proportion to the amount of Bitcoin transferred. Stacks block producers then follow the longest chain rule. This mechanism ensures that the entire history of Stacks forks is public unless the adversary launches a majority attack against the underlying Bitcoin chain. As a result, network participants can react to reorg attempts (e.g., by increasing the number of required confirmations) in order to make it far more difficult to launch profitable attacks [452].

16.3. Sidechains, Relays, and Asset Transfer

A *sidechain*, first proposed in [445], is a blockchain whose currency is "pegged" to a "main chain" but is responsible for its own consensus security, independent of the main chain. A sidechain protocol, therefore, includes a main-chain consensus mechanism, a sidechain consensus mechanism, and cross-chain communication. A sidechain may be centrally administered, federated (i.e., permissioned), or permissionless, though permissionless sidechains have known issues and are relatively untested in the real world. A sidechain may have a two-way peg, in which case sidechain assets can be transferred back to the main chain or a one-way peg, where sidechain assets permanently remain on the sidechain. Several desirable properties of pegged sidechains were described in [445]:

- Assets should be able to be transferred with minimal counterparty risk.
- An asset on a sidechain should be able to be transferred back to the main chain by the current holder of the asset and no one else, including previous holders of the sidechain asset (if there is a two-way peg).
- Asset transfers should not have failure modes that result in the loss of funds or the fraudulent creation of assets.
- Participants in the system should not need to monitor sidechains that they are not actively involved with.
- Sidechains ought to be firewalled from each other and the main chain. That is, if there is a bug or blockchain reorganization in a sidechain, asset inflation or the theft of assets should be localized to the sidechain in question.

Examples of federated sidechains include private Ethereum sidechains [453] and the Liquid sidechain on Bitcoin [454, 455]. A federated sidechain may employ various security features, such as having separate federations for managing sidechain consensus and moving assets from the sidechain to the main chain, as well as using hardware security modules for transaction and consensus signing.

A related interoperability concept is that of relays, which operate as smart contracts on one ledger that act as light clients for another ledger. The relay smart contract is supplied with block headers from the alternative chain that it needs to validate. Examples include BTC Relay, allowing Bitcoin verification on Ethereum [456]; PeaceRelay, which creates a two-way bridge between Ethereum and Ethereum Classic [457]; Dogethereum, allowing Dogecoin on Ethereum [458]; and XCLAIM, which can work with a variety of ledgers [459]. A scalability issue with this simple type of relay is that there is an increasing overhead for submitting an up-to-date block header as the length of time since the prior submission increases. All intermediate headers must be submitted and validated as well.

While most relays accept the full chain of block headers from the alternative chain, some designs are more advanced. For example, FlyClient is a more efficient way of performing light validation using only a logarithmic number of block headers [460]. Other relays use advanced cryptographic primitives, like SNARKs, to produce concise proofs of valid chain progression [461, 462]. Another proposal uses cryptoeconomic incentives to optimistically accept block headers instead of validating them and allows submitted headers to be challenged. Submitters post a bond to the relay smart contract that is forfeit if a successful challenge proves the header invalid [463].

In addition to relay smart contracts, there are relay chains that are purpose-built for interoperability, such as Polkadot [464] and Cosmos [465]. These schemes have a central blockchain that acts as a communication hub between the various interoperable blockchains within their respective networks, not unlike how sharding works (Section 15). The two projects differ in several important ways [466]:

- The security models of each project differ. In Polkadot, the interoperating blockchains inherit the security of the global relay chain. As a result, relay chain validators have the final say over state changes in the individual "shards" (called "parachains" in Polkadot parlance). In contrast, individual chains in the Cosmos network are independent and responsible for securing themselves. This makes the chains more "sovereign" by allowing them to be run independently but also requires them to bootstrap their own security.
- The governance of the respective projects differ. To become a parachain in Polkadot, one must acquire a significant quantity of Polkadot's native DOT cryptocurrency and stake them. Polkadot can only support a particular number of parachains. Cosmos lacks fixed membership rules, allowing anyone to establish a new blockchain within the network with its own custom governance process.
- Polkadot allows arbitrary message passing between participating parachains, including interoperable smart contract calls. Cosmos is particularly focused on asset transfer between chains. Architecturally, this manifests itself in the "tight coupling" of Polkadot's parachains, achieved by using fraud and data availability proofs to handle invalid blocks on parachains (see Section 15.5). Cosmos lacks this tight coupling, so users of participating blockchain A need to "trust" the relay chain used to communicate with blockchain B. In Cosmos, a malicious validator cannot corrupt a blockchain that they do not belong to. In Polkadot, on the other hand, the entire system is impacted if invalid parachain blocks are not challenged.

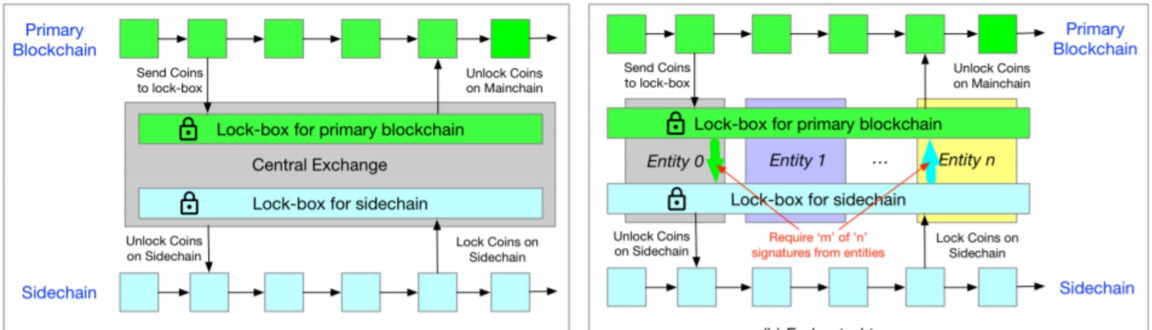
- The two projects use different consensus algorithms. Polkadot uses the GRANDPA algorithm, which scales to a potentially large number of validators [376]. The Cosmos relay chain, or hub, uses Tendermint consensus, which handles fewer validators but provides quick finality [358, 360].

16.3.1. Permissionless Sidechains

Sidechains that utilize a permissionless consensus algorithm are more challenging to design than permissioned sidechains, despite being the original motivation for the sidechain concept in the first place. As originally conceived, the state machines of sidechain and main chain would need to be capable of acting as light clients for other blockchains. In particular, they would need to be able to validate SPV proofs in order to verify that transfers from the main chain to the sidechain (and vice versa) were included and stable in the canonical chain [445].

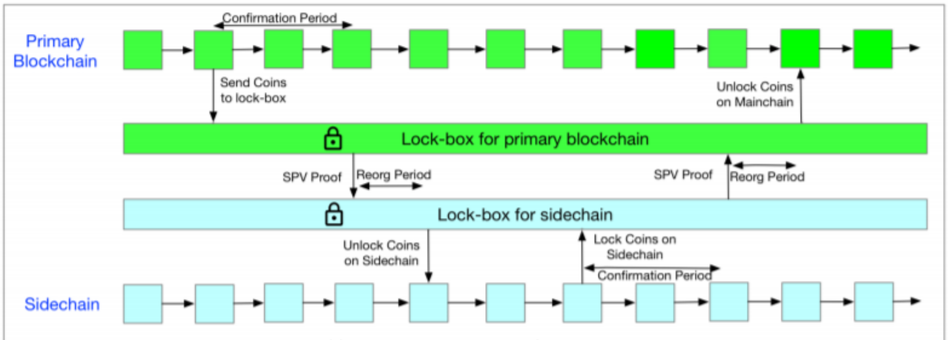
For this type of sidechain, there would be special outputs on the main chain that are designated as belonging to the sidechain. To transfer coins from the main chain to the sidechain, a user will lock their coins in one of these special outputs for a certain *confirmation period*. The confirmation period exists to allow enough proof of work to be generated to prevent denial-of-service attacks against another waiting period that follows. Once the confirmation period is over, the user can create a transaction on the sidechain that references the special main-chain output and uses an SPV proof to demonstrate that the locking was performed correctly and generate corresponding sidechain coins. Once this is done, the user must wait for a *contest period* during which the newly converted assets are unspendable on the sidechain. This contest period is used to prevent a double-spending attack vector where previously locked coins on the main chain are transferred to a non-sidechain address due to a majority hash rate attack. During the contest period, any sidechain user may publish a reorganisation or reorg proof showing that there exists a main chain with greater aggregate proof of work and where the block containing the locked output no longer exists. This invalidates the main-chain-to-sidechain transfer. This two-way pegging mechanism is shown in Figure 54, where it is contrasted with federated and centralized pegs.

There are several challenges with this sidechain mechanism. First, it leads to the risk of centralization of mining in the same way that merged mining does. Larger miners are more able to cope with the demands of validating the sidechain and are thus more likely to collect transaction fees on the sidechain. Over time, it is conceivable that the extra revenue source can drive smaller miners out of business. Second, the security model seems to rely on the ability to publish reorg proofs on the sidechain. However, an adversary capable of performing a lengthy reorg on the main chain is likely capable of doing the same on the sidechain, in which case they can censor the inclusion of a reorg proof. This kind of sidechain, therefore, may allow a majority hash rate attacker to steal all of a sidechain's coins for free. Third, SPV proofs grow linearly with the size of the chain, so main-to-sidechain and side-to-main-chain transfers will involve increasingly large transactions. Luckily, these proofs



(a) Centralized two-way peg

(b) Federated two-way peg



(c) Two-way peg based on SPV proofs

Fig. 54. Sidechain pegging methods. [467]

can be made logarithmic using modified light client protocols, such as FlyClient [460]. An alternative technology that can make proofs logarithmic is the idea of *non-interactive proofs of proofs of work* (NIPoPoWs), which use skip lists to take advantage of blocks that prove more work than necessary for the difficulty level, which occur with relatively predictable frequency [468, 469]. Unfortunately, NIPoPoWs are currently unable to handle chains where the difficulty level is variable, making them less practical than FlyClient.

Proof of stake sidechains have also been proposed [470]. In this case, instead of proving that a sufficient amount of work occurred during the confirmation period, a proof includes the set of signatures from the elected slot leaders whose length is proportional to the common prefix confirmation security parameter. One of the challenges related to bootstrapping a proof-of-stake sidechain is that it would be highly vulnerable to attack before a significant amount of stake is dedicated to it, but validators may not want to commit their stake to the sidechain while it is vulnerable. This is addressed by the idea of merged-staking, where stakeholders from the main chain can signal their awareness of a sidechain in order to be eligible for block creation on that sidechain but without needing to transfer the stake itself.

Another proposal for proof-of-work sidechains is called Drivechains [447, 448]. A Drivechain is like a "plugin" for a main chain, and a main-chain full node is a required component of running a sidechain full node. In the context of Bitcoin's state machine rules, all coins in a particular Drivechain are locked in an "anyone can spend" address on the Bitcoin chain, and a new soft-forked rule is added to prevent these coins from being spent on the Bitcoin chain unless additional conditions are met.

Drivechains replace the requirement of submitting chains of block headers for each transaction with a more explicit form of miner voting. Transfers from the main chain to the sidechain no longer require a lengthy confirmation period and can be done nearly instantly upon deposit in the "anyone can spend" address. On the other hand, sidechain-to-main-chain transfers use a much slower mechanism. When sidechain users want to withdraw their coins back to the main chain, they create a sidechain withdrawal transaction that burns the sidechain coins. Over an extended period of time, these withdrawal transactions accumulate on the sidechain and are eventually aggregated into a large main-chain transaction from the locked "anyone can spend" address to the withdrawal addresses specified in the sidechain withdrawal transactions. The transaction ID of this aggregated withdrawal transaction can then be embedded in the coinbase transaction on the main chain and represents a withdrawal attempt. For some pre-specified number of blocks following the inclusion of this withdrawal attempt, miners vote on whether the withdrawal is valid. If the miner vote passes a threshold in favor of the withdrawal, the actual withdrawal transaction is included in a block. Due to the lengthy withdrawal period, users may tend to perform atomic swaps with main-chain coins rather than withdraw directly.

This security model allows a majority coalition of main-chain miners to steal the full contents of a sidechain by creating and voting to approve any withdrawal transaction of their choosing. Because the main chain is unaware of the sidechain state, users of the sidechain may be required to organize themselves into a user-activated soft fork that prevents this theft. Realistically, this would only be plausible with relatively popular sidechains.

A final piece of the Drivechain puzzle is the use of *blind merged mining* to reduce the centralizing impact that other permissionless sidechains may have [446]. The underlying centralization concern is that the sidechain fees are only available to those entities with the resources to run a fully validating sidechain node. Blind merged mining allows main-chain miners to opt out of running a sidechain node but without the opportunity cost of losing sidechain fees. It does this by providing a mechanism for main-chain miners to "sell" the service of finding a sidechain block in a trustless way. Miners on the sidechain know that a valid sidechain block with hash H is worth, say, X sidechain coins in transaction fees and are thus willing to pay up to (approximately) X main-chain coins to have H included in the main chain's coinbase transaction (and thus acquire X sidechain coins themselves). A main-chain miner can be unaware of the sidechain and still accept payment of X main-chain coins to include H in a block they are constructing at no cost to themselves. Should H be invalid according to the sidechain rules, the sidechain can simply ignore it, and the main-chain miner keeps the main-chain payment anyway. The sidechain miner can keep a small amount for themselves, but this should create a competitive market with main-chain bribes close to X.

17. Networking

In any distributed system, processes are required to communicate over a network. In the majority of permissioned systems, every node will have direct, point-to-point communication channels with every other node (see Section 8 for some examples of exceptions). Permissionless networks, on the other hand, have a more complex design space.

17.1. Networking for Permissionless Systems

Neudecker and Hartenstein identify five goals for the networking layer of a permissionless system: 1) performance, 2) low cost of participation, 3) anonymity, 4) resistance to denial-of-service attacks, and 5) topology hiding [471]. These goals sometimes conflict with each other. For example, high performance and resistance to denial-of-service attacks tend to require more data to be transmitted, while anonymity, low cost, and topology hiding tend to require transmitting less data over the network. All five of these goals have implications for system security. For example, if the performance of block dissemination is not extremely fast in Nakamoto Consensus, the security bound decreases, as discussed in Section 10.2.1. Low cost of participation is critical to permissionless systems in general lest full node ownership not be well distributed (see Section 3.2).

Network-layer adversaries may engage in passive or active attacks to achieve certain goals. An adversary may want to deanonymize users of the system by linking network information to application-layer information. For example, the adversary may attempt to discern which IP address a particular transaction was first broadcast from. Alternatively, the adversary may try to exploit the network to deny service to certain nodes in order to provide the attacker with a monetary reward. For example, an attacker could induce a network partition to increase γ for selfish mining purposes (see Section 9.4), and fully isolating a node through an eclipse attack can increase the chance of a successful double-spend. These attacks all benefit from knowledge of the network topology, so an adversary may try to map the topology as an initial phase of their attack.

Methods for topology inference have been studied for Bitcoin [472], Ethereum [473], Zcash [474], Monero [475], and probably other systems. Various methods of connecting IP addresses to transactions have been suggested for the Bitcoin network (similar methods would work for comparable systems), some of which find or exploit knowledge of the topology [476–478]. Networking side-channels have also been found to deanonymize users of private cryptocurrency systems Zcash and Monero [479].

Network-layer attackers may take advantage of nuances in the networking protocols of permissionless systems in order to force a victim node to connect to the network solely through attacker-controlled nodes, thus allowing an attacker to control the victim's view of the blockchain. Specific eclipse attacks have been found against both the Bitcoin [480] and Ethereum networks [481, 482]. A security requirement for a node on a permissionless network is to have at least one honest peer connection, but eclipse attacks can violate this assumption and make targeted double-spending attacks much easier.

An attacker may also attempt to cause more significant network partitions. In practice, Bitcoin nodes are concentrated among relatively few autonomous systems, so an adversary could cause network partitions by performing BGP hijacking of relatively few IP prefixes [483, 484]. On the other hand, if a significant number of mining pool nodes are hidden behind Tor, BGP hijacking can be made significantly more difficult [485]. BGP hijacking is easily detected, but far stealthier network partitioning attacks have also been proposed [486]. Due to there being IP addresses that host nodes from multiple cryptocurrency networks, it may be possible to attack several networks at once by attacking a small subset of nodes [487]. There are designs for relay networks that are resistant to BGP hijacking attacks, such as SABRE [488]. Additionally, blocks can be transmitted over satellite or radio networks in case of network partitions.

More esoteric attacks are also possible, such as *timejacking* [489]. Some systems, such as Bitcoin, have nodes adjust their clocks based on the median timestamp reported to them by their peers. This allows a Sybil attack variant where malicious nodes connect, report incorrect times, and move the clock of a victim node. This can get them to do things like temporarily believe that a block is invalid and cause temporary network partitions. In tandem with a software bug, this can also cause permanent chain splits that require manual intervention to fix [490].

17.1.1. Peer Discovery

The first thing that a new node needs to do upon startup is find a set of reachable IP addresses in order to create outgoing connections to peers. Usually, the software for running a node includes some hard-coded "seed" nodes that are DNS servers run by respected community members. This is inevitably a more centralized process than is desirable, as malicious seed nodes can eclipse a victim node by providing only IP addresses under their control (if the user contacts only a single seed node). A malicious seed node can also use their position to strategically impact the network topology. Finally, if the seed nodes are unavailable, new nodes are unable to start, which may make them a target for denial-of-service attacks.

To mitigate these risks, there should ideally be a relatively large number of seed nodes being operated and available. When a new node starts up for the first time, it should contact multiple different seed nodes and establish outgoing connections using IP addresses supplied by each of them. Topology hiding can be improved at the expense of increased cost of participation if clients only connect to a small portion of the IP addresses supplied by the seeds.

Once a node has an established connection, it can also discover new peers from the existing ones, either by asking for more IP addresses or receiving them unsolicited. Here, there is a tension between the quality of the connection and topology hiding. When IP addresses are announced, it provides information that can be used as part of an eclipse attack or to infer the network topology, especially if nodes announced their neighbors. On the other hand, announced IP addresses should be reachable nodes, which suggest strategies such as announcing the IP addresses of recent or current connections. Because nodes frequently leave and rejoin the network, there is a risk of announcing IP addresses that are old and unreachable.

Accounting for this node churn is important. The overwhelming majority of nodes will leave and rejoin the network multiple times over the course of a couple of months, the average exceeds four churns per node per day, and churn increases block propagation time by weakening the performance of compact blocks (discussed in Section 10.2.1) [491]. Considerable churn has been observed on the Ethereum network as well [492]. Luckily, large networks with more than 4,000 nodes are fairly resistant to the negative effects of churn [493].

17.1.2. Neighbor Selection

Peer discovery provides a node with a set of IP addresses to establish outgoing connections to. With this set, the node must decide which specific peers to connect to, as well as how to handle incoming connection requests. Given a list of IP addresses, a node can either try connecting to random ones or use their knowledge about the peers to inform the decision. This knowledge can include information based on the IP address itself, such as the AS or region where the IP address is located. Alternatively, it can use information based on past experience with the peer or information from trusted external sources. A Bitcoin node, for example, limits the number of outgoing connections it establishes within a given IP address range. This forces adversaries to control a larger and more diverse set of IP addresses in order to perform an eclipse attack. Bitcoin nodes also utilize past observations from their own connections to block peers that misbehave.

Another component of the neighbor selection policy is deciding how many outgoing and incoming connections to establish. A larger number of connections can improve block propagation speed and make the network more robust to eclipse and partitioning attacks. On the other hand, more connections increase the cost of participation. Outgoing connections are more trustworthy than incoming ones because even weak adversaries can establish a significant number of incoming connections to an honest peer. As a result, denial-of-service resistance increases to a greater degree from more outgoing connections than additional incoming ones. That said, a large number of incoming connection slots is desirable so that other peers can establish outgoing connections and to prevent an adversary from using up the available incoming connection slots.

The node software must also decide whether to maintain established connections for as long as possible or to strategically disconnect from some neighbors and replace them with others. Maintaining connections for longer makes it easier for an adversary to infer the network topology but more challenging to exploit it using their own Sybil nodes. It is also possible for the neighbor selection protocol to strategically generate certain network topologies. For instance, creating clusters of nodes based on geographic proximity can speed up information propagation but makes it easier to infer the network topology and launch various attacks.

Finally, nodes can monitor their own connections for performance and security and disconnect from peers that send too much or too little information, are slow to respond, or otherwise behave unexpectedly. While this can improve resistance to attacks, adversaries can occasionally take advantage of these mechanisms to aid in other attacks. For example, an attack on the Bitcoin network was proposed where nodes connecting over Tor can be eclipsed by taking advantage of peer monitoring [494].

17.1.3. Communication Strategy

While peer discovery and neighbor selection determine the network structure, nodes must also have a communication strategy to determine how best to disseminate information. Messages may be treated differently based on their content or the context they occur in. For example, in Bitcoin, a received block is immediately relayed (after being verified), whereas transactions are relayed with a random delay. Similarly, a node may relay transactions differently when created by the user of the node itself as opposed to transactions received from the network. As a final example, nodes may not participate in transaction relay at all when they are still performing their initial block download to synchronize with the network. Using message content and context in this way can help balance competing priorities.

One dimension of the communication strategy is whether messages should be pushed in an unsolicited fashion or whether messages should be announced first and only sent directly if requested by the peer. Pushing messages results in them being disseminated across the network more quickly than the announce-and-request method because information is sent in half a round trip instead of 1.5 round trips. On the other hand, the announce-and-request method uses much less bandwidth than the unsolicited push by avoiding duplicate messages. Nodes must also decide whether to flood a message to all of its neighbors or gossip the message to a (possibly random) subset of connections. Flooding uses more bandwidth than gossip but is also more robust against attacks. A node might, for instance, pick a few peers to push unsolicited blocks to in order to speed up block propagation while following announce-and-request for blocks sent to other peers (and transactions sent to all peers) in order to keep bandwidth consumption modest.

Nodes must also determine when to send messages to their peers. They can, for instance, add a random delay before forwarding messages, which can improve topology hiding and anonymity while slowing down performance. They can also choose to aggregate several messages together to send simultaneously, which reduces the bandwidth overhead. For instance, a node might forward five transactions at a time. This has other potential benefits, such as making the code cleaner by maintaining only a single outgoing message queue per connection. Allowing messages to accumulate has an unclear impact on propagation speed but can improve it on average if new outgoing messages are added to a queue that was already scheduled to be sent (assuming they would otherwise be sent with a random delay). Message accumulation also has an unclear impact on anonymity and topology hiding. Message transmission times depend on information about received messages that an attacker may not be privy to. On the other hand, it can open up new attack vectors, such as inferring when a peer received a certain message from another peer using information from the aggregated set of messages.

18. State Machines

Distributed ledger systems utilize consensus in order to execute a state machine, where a set of servers execute commands on behalf of clients in an agreed-upon order to modify the state of the system. The state machine is the set of rules that validating servers enforce while transforming the state based on these client-submitted commands, called transactions. While a wide variety of rules are possible, some types of rules appear frequently. For instance, a transaction that modifies a piece of the state of the system should include a valid signature from a client that is authorized to act on that portion of the state. Other common rules include establishing a maximum block size that block proposers can create, requiring a particular syntax for blocks and transactions, and ensuring that all transactions included in a block are committed to in the block header.

The idea of state machines was introduced in Section 2.8. There are two primary models that distributed ledgers use to organize and manage their state: the UTXO model and the account model, discussed in detail in Section 2.8.1. Some unique challenges with respect to changing the rules of distributed ledgers were discussed in Section 2.8.2. This section will provide some additional details regarding state machine design. A variety of alternative state machines with different properties are possible. In addition, "off-chain" or "second layer" systems that take advantage of the properties of the underlying state machine to improve scalability are introduced.

18.1. Virtual Machine Design

The majority of distributed ledger systems include a runtime environment or virtual machine that executes platform-independent code in a low-level programming language (exceptions exist, such as Monero). Transactions act as miniature programs or function calls that modify a portion of the state of the ledger by executing a series of *opcodes*, or instructions. Some systems – like Bitcoin – use a set of opcodes with limited functionality, while others – like Ethereum – use a Turing-complete instruction set capable of executing any general program. In many cases, higher-level languages exist that are friendlier to the programmer and compile down to bytecode made up of virtual machine-specific opcodes.

Bitcoin uses the Bitcoin Script language, a Forth-like, stack-based scripting system with no loops. The original implementation had many more opcodes, but a large number of them were deprecated early on in Bitcoin's history due to fears of denial of service and other attacks. The instruction set includes a number of "no operation" opcodes, `OP_NOP`, which allow new opcodes to be added via soft fork. In Bitcoin's UTXO model, outputs contain a *scriptPubKey* field that includes an encumbrance, or a condition to satisfy in order to spend the output. When an input spends an output, it uses the input's *scriptSig* field to include data and opcodes that satisfy the spending condition in the *scriptPubKey*. To validate an input being spent, an empty stack is created, and the program to execute consists of the *scriptSig* prepended to the *scriptPubKey* of the output being spent. This combined script is executed from left to right and is considered valid if nothing triggers failure during the script execution and the top stack item when the script completes is `True` (or non-zero).

One of the most common, standard transaction types in Bitcoin is the pay-to-public-key-hash (P2PKH) transaction, which transfers some bitcoin from one address to another (where an address is a hash of a public key). In this case, the *scriptPubKey* will look like the following, where bracketed items are data to be pushed to the stack: `OP_DUP OP_HASH160 <Owner's address> OP_EQUALVERIFY OP_CHECKSIG`. The corresponding *scriptSig* consists of two push operations: `<Owner's signature> <Owner's public key>`. As a result, the combined script to be executed is the following: `<Owner's signature> <Owner's public key> OP_DUP OP_HASH160 <Owner's address> OP_EQUALVERIFY OP_CHECKSIG`.

Execution proceeds from left to right, so the first thing that happens is that the signature is pushed to the stack, and then the owner's public key is pushed to the stack above it. The `OP_DUP` operation duplicates the top item on the stack (in this case, the public key). Next, the `OP_HASH160` opcode takes the public key at the top of the stack and replaces it with the RIPEMD-160 hash of the SHA-256 hash of the public key. The owner's address is then pushed to the top of the stack. From top to bottom, the stack now consists of the address supplied in the output, the hash of the public key given in the input, the public key itself, and the signature from the input. The `OP_EQUALVERIFY` opcode checks that the top two items on the stack are equal and fails if they are not, in which case it ensures that the public key supplied in the *scriptSig* is a preimage for the address given in the *scriptPubKey*. Assuming that they match, the top two stack items are removed, leaving just the public key on top of the signature. Finally, `OP_CHECKSIG` takes the public key and signature, pops them off the stack, and pushes `True` if the signature is valid.

To check whether the signature is valid, the execution environment needs to know what message is being signed. To this end, the signature includes a one byte *SIGHASH flag* that determines which portions of the transaction are used to form the message. This provides some flexibility for users, so that only some parts of the transaction are signed, allowing others to modify the transaction if desired. For instance, it may be desirable to have the message only include a single output, allowing other users to contribute additional inputs to the transaction so long as the specified output is included too.

In contrast to Bitcoin, Ethereum uses the Turing-complete Ethereum Virtual Machine

(EVM) as its execution environment and uses the account model rather than the UTXO model. The EVM is capable of executing any deterministic program but subject to the EVM's *gas limit*. Gas is a unit of measurement that is intended to correspond to the computational effort required to execute an opcode, so each opcode has a particular gas cost. The gas limit is analogous to a maximum block size but for computational effort instead of space. There are two types of accounts in Ethereum: contract accounts and externally owned accounts (EOAs). EOAs are typical user accounts controlled by a private key that allows the user to transfer the ether currency or interact with contract accounts. Special transactions are used to deploy smart contracts by creating contract accounts that include the smart contract's bytecode. These contract accounts are then controlled by the EVM bytecode when an EOA triggers a function call on the smart contract.

The EVM's global state is a mapping between addresses and the associated account's state. Each account has an ether balance, a nonce, a hash of the code deployed to that address, and a hash of the account's storage. There is also a machine state that includes a program counter pointing to the next instruction to execute, the remaining gas available, a stack, and memory. To invoke a smart contract, a user signs a transaction where the *recipient* field contains the smart contract address, and the *data* field contains information on the function to be called and arguments to pass to it. A transaction's execution may fail and throw an exception, in which case gas fees are still charged to the account and given to the transaction's miner, but any state transitions the transaction would have caused are reverted.

It is extremely challenging for developers to write smart contracts directly as low-level bytecode. Instead, developers will typically use a high-level language, such as Solidity or Vyper, and compile it down to EVM bytecode before deploying. Other high-level languages have been developed for other distributed ledger platforms, such as the Move language created for the Diem blockchain. Move was designed to handle the problem of conservation, or ensuring that fund transfers preserve the total monetary supply in the system [495]. Solidity handles this naturally by associating each account with a balance that can only be modified by special instructions. For tokens built on Ethereum, the EVM cannot maintain this supply conservation property without encoding it in the token smart contract itself, whereas the Move language maintains this property even for custom tokens.

A number of projects use WebAssembly (WASM) as their runtime environment for smart contracts. Due to its support on most modern web browsers, distributed ledger clients will be able to run in the browser more easily. WASM instructions can be directly mapped to machine instructions, so it should be highly performant. In the future, Ethereum will likely migrate from the EVM to eWASM, or a restricted subset of WebAssembly designed for Ethereum. Specifically, eWASM is the same as WASM but with floating point non-determinism removed, gas metering added, and an Ethereum-specific interface [496]. While eWASM is expected to ultimately improve performance, current implementations have variable but relatively poor execution speeds [497].

A final point on Turing-completeness is in order. An interesting side-effect of having a

Turing-complete instruction set with gas metering is that it makes it unsafe to conduct some kinds of soft forks due to opening up a denial-of-service vector [498]. On the bright side, this also prevents some malicious soft forks, such as smart contract censorship. Consider a soft fork that made any transaction that operates on a particular contract invalid. Once the soft fork is deployed, an attacker can broadcast many transactions to the network that perform a variety of challenging computations before invoking the censored smart contract. Miners that run the soft fork code would have to execute these transactions before finding them invalid, but due to the soft fork, they would not be able to claim gas fees as compensation, nor would the attacker pay fees. Since this costs the attacker nothing, they can amplify the attack by setting high gas prices, encouraging miners to waste more computational resources. Static analysis could be used to see whether the transaction interacts with the censored address, but this address could be obfuscated, and static analysis is itself computationally intensive. Turing-completeness implies that, due to the halting problem, transactions must be executed in order to determine how the computation will unfold.

A different approach is taken in the Mimblewimble protocol, where the scripting language is removed entirely [499–501]. Mimblewimble's design prioritizes improved scalability for newly joining nodes while also providing a privacy boost for users. When a node joins a Mimblewimble network, the bandwidth and computational effort required to synchronize with the network is roughly proportional to the size of the UTXO set, whereas the effort is proportional to the entire transaction history for Bitcoin and most other ledgers. This is accomplished by allowing *transaction cut-through*. Consider a ledger following the UTXO model. If a transaction spends output TXO_1 and creates TXO_2, and a second transaction spends TXO_2 and creates TXO_3, this is equivalent to a single alternative cut-through transaction that spends TXO_1 and creates TXO_3. In Bitcoin, cut-through is impossible once transactions are included in the ledger, but the Mimblewimble ledger is ultimately a single aggregated transaction with a set of outputs equivalent to the full UTXO set. With no scripting language, Mimblewimble lacks a virtual machine, but it still performs state machine replication when combined with a consensus algorithm.

18.1.1. Concurrency in Smart Contracts

In state machine replication, where all transactions are totally ordered, it is natural to think in terms of sequential execution of transactions. However, one of the best ways to improve performance in computing is to exploit parallelism to make better use of available CPU cores. The primary challenge in adding concurrency to state machine execution is that the runtime environment requires determinism to ensure that validators remain in agreement with one another. It may nevertheless be worthwhile to attempt to solve this problem since some estimates suggest that if all available concurrency opportunities in Ethereum were exploited, there would be a factor of six improvement in execution speed [502]. A variety of approaches to adding concurrency to smart contracts have been explored [503–509]. Solana, for example, is a prominent smart contract platform where transactions must specify in advance the portions of system state that are accessed, which allows validators to execute non-conflicting transactions in parallel [510].

An early approach used a locking mechanism to divide transactions across several threads of execution [503]. Whenever a transaction attempts to access a portion of the system's state, the thread attempts to acquire a lock for it and will not execute the transaction without having a lock on the relevant state. Because this is not a deterministic process, a miner needs to keep track of the history of which threads acquired which locks. This history creates a happens-before graph of the executed transactions, which must be transmitted to validators as a part of the block. Validators can use this graph to spawn the required threads and assign transactions to them.

Another approach utilizes speculative execution in order to achieve concurrency in a lock-free manner [504]. First, all transactions are executed concurrently, assuming that there are no conflicts. When conflicts do occur, the transaction is aborted and put into a separate "bin" for sequential transactions. Once the speculative concurrent execution of all transactions has completed, the sequential transactions are executed. As a result, when a transaction causes a conflict, it must be executed twice, introducing some overhead.

A third approach takes advantage of software transactional memory systems (STMs) [506, 507]. In contrast with the use of locks, transactional memory assumes that simultaneous access to state will not cause conflict; that is, it is optimistic. Therefore, threads do not need to wait for each other in order to access the same portion of the state. During transaction execution, miners attempt to update the state but do not fully commit these changes until later. When a state change is about to be committed, the runtime environment checks for conflicts based on prior commits, and the changes are reverted in case of conflict. This process is non-deterministic in the same way as using locks, so the miner generates a happens-before graph to transmit to validators in a similar way. This method results in a lot less waiting time when multiple transactions access the same piece of state but in ways that do not cause conflicts.

One can also parallelize existing ledger systems by using some basic static analysis. For example, [505] defines the notion of *strongly swappable* transactions: transactions TX_1 and TX_2 are strongly swappable if the state variables that each transaction accesses are disjoint. To exploit this, any method of static analysis can be used to over-approximate the portions of state accessed in a transaction. For UTXO-based ledgers, such as Bitcoin, a simple check on the inputs and outputs is sufficient. Strong swappability then implies a partial ordering of transactions that validators can deduce themselves and that is guaranteed to be equivalent to a serial execution of transactions.

Finally, [508] describes a clever solution that utilizes a variant of locking, concurrency delegation, and static analysis. Each state variable is augmented with a taint value. A state variable that has not been accessed during the current execution is considered untainted, but once a thread attempts to read or write to it, it is considered tainted with a value that identifies the thread. This is a relaxation of the idea of locking such that long wait times are avoided. Instead, if a thread attempts to access a tainted variable, the execution immediately stops. The thread then forwards the conflicting transaction to the thread that tainted the variable, which is more likely to be able to execute it. After the transaction is delegated in this way, it may still not be able to be executed, in which case it is added to a queue of sequential transactions to be executed later.

This delegation process requires that transactions are initially distributed to threads in some way, ideally while minimizing conflicts. To this end, transactions are analyzed based on static information to provide "hints" without needing to execute the transactions. For example, the sending address in a transaction constitutes a good hint. A programmer can then annotate transactions with these hints, which are beneficial despite being imperfect. A primary thread uses this information to distribute transactions across worker threads, who use tainting and delegation as described above. If a tainted transaction cannot be executed by the thread it was delegated to, the worker thread sends it back to the primary. Workers let the primary thread know that they have completed execution, at which point the workers are shut down, and the sequential queue is executed. Finally, the primary thread labels transactions based on which worker thread they were executed by or whether they were sequential, providing that information to validators and allowing deterministic execution.

18.1.2. Zero-Knowledge Proofs and Verifiable Computation

A common technique in distributed ledger protocols is to employ zero-knowledge proofs, frequently zk-SNARKS, in order to prove that some computation was performed correctly while revealing no additional information. Depending on the exact techniques used, this can enhance privacy as well as improve the scalability of distributed ledgers by having validators verify short proofs of computation instead of performing all computations themselves.

An early application of this is the Zerocash protocol, which has been deployed in the Zcash network [511]. In Zcash, zk-SNARKS are used to create "shielded" transactions that hide the transaction's inputs, outputs, and amount. The zk-SNARK proves to the rest of the network that the transaction adhered to the state machine's rules. The proof demonstrates that the amount of Zcash transmitted in the inputs is equal to the amount of Zcash in the outputs less a fee and that the spender possesses the requisite private key to spend funds. Similar techniques can be used to add privacy to smart contracts on Ethereum or other Turing-complete platforms. Zether, for example, uses a different form of zero-knowledge proof (called Bulletproofs) to hide transaction amounts while performing fund transfers to and from smart contracts [512]. This enables things like sealed-bid auction smart contracts.

Other schemes make the smart contract computations themselves more private rather than just payments [513–515]. For example, the Hawk system can be used to hide the inputs to smart contracts from everyone except a special party called the manager, which may be implemented via multi-party computation [513]. Zexe hides not just the inputs and outputs of a function to be executed but also the function itself as well as the internal state of the smart contract that executes it [514]. To see why this is valuable, one can imagine that every token on Ethereum is built with a Zerocash-like scheme that hides the transaction details of token transfers. In this case, transactions would still leak which particular tokens were involved in a transaction. An eavesdropper may not know how much of a token was transferred, but they could still deduce the token smart contract that was used. In Zexe, a shared execution environment is created in which transactions reveal no information about the offline computations performed for a smart contract, are unlinkable to other transactions by the same user and/or the same type of computation, and can be verified in constant time regardless of the complexity of the computation. Users perform the computations themselves (or, in an extension to the protocol, delegate the computations to someone else) on plaintext inputs, encrypt the inputs and outputs to the computation, and combine it with a zk-SNARK before submitting the transaction to the network. Validators only need to verify that the zk-SNARKs included in the transactions are valid but do not need to re-execute the computations.

Another approach, dubbed smartFHE, uses fully homomorphic encryption to allow smart contracts where users maintain privacy over their inputs and outputs to the contract function [515]. That is, it allows computation to be performed on encrypted data supplied by one or more users, where the results of the computation remain private to those users as well. Fully homomorphic encryption allows users to submit to a smart contract a list of encrypted inputs to a function to execute as well as a zero-knowledge proof that the submitted ciphertexts are well-formed. After verifying the proof, miners will execute the requested function calls on the ciphertexts. In contrast to Zexe, the private computations in smartFHE are performed on-chain, and the smart contract's code is public. smartFHE does not provide anonymity to users interacting with the smart contracts.

18.1.3. Delegating Execution

Most distributed ledgers require all participants to execute every transaction. Combined with a consensus algorithm that guarantees agreement over a total ordering of these transactions, this is a natural way to achieve state machine replication. However, requiring that every node executes every transaction hinders the scalability of the network and reduces the privacy of the computation. It would be better if only a subset of participants were required to execute transactions rather than every node on the network.

TrueBit is a scheme that uses smart contracts to outsource computation and provides economic incentives such that rational participants will execute the computation correctly [516]. As a result, network validators need not execute these computations, while the complexity of the computations can be significantly greater than the base layer's gas limit would typically allow. A user who would like a computation executed deposits some funds to pay for the result. Anyone can act as a solver or verifier in the system by submitting a deposit to the contract, though solvers are matched to computational tasks at random. To provide proper incentives for verifiers in TrueBit, potential verifiers need to believe that there is a real chance of finding a flawed computation from a solver. As a result, TrueBit periodically requires incorrect solutions to be submitted. Once a solver is randomly assigned to a task, they calculate both a correct and incorrect result to the computation and send commitments to both to the TrueBit contract. The contract determines whether a "forced error" should occur for this computation, in which case the solver opens the commitment to the incorrect result (otherwise, they open the correct result). Verifiers then check the solution and issue a challenge if they disagree with the solver. The verifier receives a large payment if the solution is erroneous and the forced error regime is in effect. The system will accept the result as accurate if no verifier challenges it, but if a challenge exists, a *verification game* is played.

Assume that the requested computation runs in t time steps and requires at most s bits of state at any given point in the computation. Both the solver and the challenger create a mapping of each time step of the computation to its internal state at that time. Let $c > 1$ be a parameter of the verification game that determines a trade-off between the number of game rounds and the amount of communication required with the underlying ledger.

The verification game runs a loop that attempts to determine where in the computation the discrepancy arises. At the start of the loop, the solver picks c equally spaced state configurations based on the current range of the disputed computation. For each of these c configurations, the solver creates a Merkle tree with s leaves that correspond to each bit of the internal state and publishes the roots to the TrueBit contract. The challenger submits $i \leq c$ to the contract, where i is the first time step in the list where they disagree with the state. The TrueBit contract verifies that c Merkle roots have been submitted and that $1 \leq i \leq c$ and has the relevant party lose if a check fails. The loop begins again but only using the state configurations between the $(i-1)$-th and i-th configurations from the prior round of the loop. Eventually, this loop converges to the first disputed step over the whole computation, and this step is then verified by the smart contract. If this is the e-th time step, the solver will submit to the contract paths from time $e-1$ and e's Merkle roots to the leaves that contain the relevant Turing-machine variables needed to check the one computational step. The winner of the verification game is determined by whether these paths are valid and the computation step was executed properly.

Another technique, employed in Hyperledger Fabric, is to decouple the ordering of transactions and their execution [517]. For most ledgers, a consensus algorithm first orders batches of transactions that are then executed serially, updating the state when execution completes. In contrast, Fabric uses the *execute-order-validate* (EOV) paradigm, where transactions are first executed, then ordered by the consensus algorithm, and finally validated for consistency to remove conflicts before updating the state. These approaches are compared in Figure 55. Because potentially invalid transactions will be ordered, throughput can be harmed by the inclusion of duplicate or conflicting transactions [518]. To mitigate the performance impact of this, client authentication and other techniques may be used.

Because not all parties need to care about every smart contract on the system, each contract in Fabric specifies a set of *endorsers* that execute the transactions and an endorsement

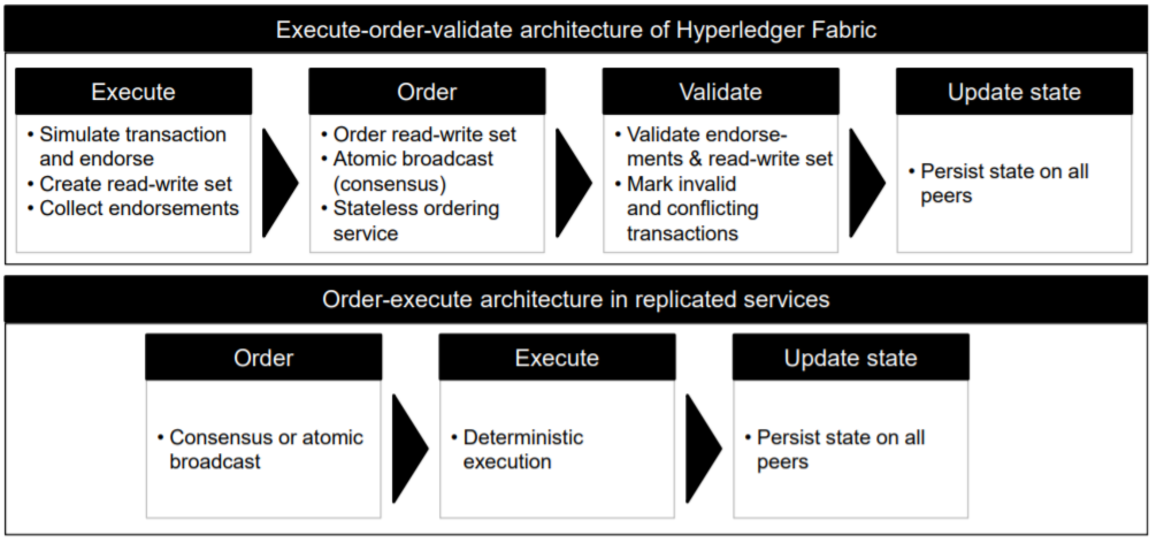


Fig. 55. Comparison between execute-order-validate and order-execute. [519]

policy that provides the threshold that must agree or any specific peers to trust. In the execution phase, clients sign transactions and send them to the relevant endorsers. The endorsers create a *read set* and *write set* of the pieces of state that were read from or written to by the transaction and reply to the client with a signed endorsement of these sets and the result. Clients collect endorsements until the endorsement policy is met. The client bundles up these endorsements and creates a signed transaction to send to the ordering service, which uses a consensus algorithm to totally order transactions. In the validation phase, replicas verify that transactions meet their endorsement policies and are marked invalid if they fail. Replicas then sequentially check transactions for conflicts in their read and write sets and are marked as invalid if conflicts arise. Finally, the state is updated by the valid transactions, the block is committed, and clients are notified.

Arbitrum is a mechanism that allows a smart contract to be defined as a virtual machine where computation is performed off-chain, and both the code and the data for the smart contract remain off-chain for improved privacy [520]. When a contract is deployed, the creator selects a set of VM managers who are in charge of executing the code. As long as one VM manager is honest, security is maintained. Transactions are sent to the VM managers, who execute them and sign the result. If there is unanimous agreement among VM managers, their signatures are published to the underlying ledger, and replicas are only required to verify the signatures instead of performing the requested computation. If parties behave irrationally, a TrueBit-like verification game is played that resolves disputes by examining the execution of a single instruction.

ACE is a scheme that is intended to allow highly complex contracts that may take minutes to execute on top of an underlying ledger, such that computations themselves occur off-chain and asynchronously [521]. As with Arbitrum, contract issuers appoint a set of service providers to execute the code. However, ACE allows flexible thresholds instead of requiring unanimous agreement and allows different smart contracts to safely interact with each other in a composable way. In ACE, blocks include separate ordering and result sections. Miners will pre-order transactions, and service providers execute transactions from this section off-chain when their contracts are called. Service providers sign and broadcast the results of the execution over the network and create a special state change transaction. If a sufficient threshold of service providers signs off on the state change, miners include it in the results section of a block, thereby committing the transaction. By having each execution result reference the preceding contract state, consistency can be maintained even when the threshold is less than half of the designated service providers. A similar scheme can also be implemented on Bitcoin and other networks with less expressive scripting support [522].

A different approach employed in LazyLedger is to have the blockchain order transactions and make them available but shift the burden of executing and validating transactions to the clients who care about them [523]. For consensus participants, verifying a block consists solely of verifying data availability, either by downloading the whole block or by using probabilistic techniques like those discussed in Section 15.5. The LazyLedger blockchain just stores messages for smart contracts, but the contract logic itself is off-chain, can be written in any language or environment, and can be changed without a hard fork. This architecture allows the users of a smart contract to ignore messages related to other contracts and only execute or validate state transitions that they care about. While this can dramatically improve the scalability of computation, there is a risk of denial-of-service attacks on smart contracts if malicious clients create many invalid transactions, so LazyLedger may work better in a permissioned system. Another limitation is that it is challenging to construct light clients for smart contracts using this architecture.

18.2. Layer 2 Protocols

Generally speaking, distributed ledgers scale somewhat poorly because every single node on the network is required to process every transaction. This consumes large amounts of storage and bandwidth, and verifying a large number of signatures can be taxing on a CPU. It is preferable for transactions to be processed locally by the parties involved rather than by every single node. This is where "layer 2" techniques can be useful. Smart contracts can rely on a small amount of global state in order to execute many transactions "off-chain," using the ledger itself only for settlement and dispute resolution.

18.2.1. Payment and State Channels

The most basic second layer scaling technology is the *payment channel*. Once set up, the participants can send a potentially limitless number of transactions back and forth between each other without needing to touch the underlying ledger except for final settlement. Instead of issuing a separate transaction for each morning coffee for a month, a customer could open a payment channel with the coffee shop on-chain at the beginning of the month, purchase each coffee off-chain, and close the channel at the end of the month. In this case, instead of having 30 different transactions processed globally, only two are required. The same idea can be applied to smart contracts in general rather than just payments, in which case it is called a *state channel*.

A variety of payment channel constructions exist [524, 525]. This section describes the basics of the Lightning Network (LN) payment channel construction [526, 527]. An LN channel consists of the following transactions (where the notation is from [524]):

- **Funding**: This transaction opens the channel and deposits funds into it, similar to a prepaid debit card. It is signed by both parties and creates a 2-of-2 multisig UTXO.
- **Commitment**: These transactions spend from the multisig output and thus require signatures from both channel participants, who receive their counterparty's signature in advance of usage. These transactions have two outputs: the first sends coins to the broadcaster, and the second sends coins to the counterparty. The counterparty can spend the second output by signing it. The first output has two potential spending conditions:
    1. A signature from the broadcaster plus a *relative lock time* such that the commitment transaction has depth λ in the ledger before being spendable
    2. A signature from the counterparty plus a preimage $S_{X,j}$ of revocation hash $h_{X,j} = H(S_{X,j})$
- **Revocable Delivery**: These transactions send coins to their broadcaster and require the signature of the broadcaster and for the commitment transaction they spend from to have a depth of λ blocks in the ledger.
- **Delivery**: These transactions immediately send the counterparty their share of coins when signed by the counterparty that did not broadcast their commitment transaction.
- **Breach Remedy**: These transactions can only be broadcast after a revoked commitment transaction is included on-chain. They require a signature from the counterparty that did not broadcast the revoked commitment transaction, as well as the preimage $S_{X,j}$ of the revocation hash from the commitment transaction. The breach remedy transaction has no relative lock time, so the counterparty can immediately take all of the channel's coins.

A channel can be closed either cooperatively or unilaterally. If both parties are online and cooperative, they can simply agree upon a transaction that sends each party their correct share. There is also a dispute process for when one of the parties is not online: either party can broadcast their most recent commitment transaction and revocable delivery transaction to recover their funds, while the counterparty then broadcasts their delivery transaction to receive their own share. If either party publishes an outdated commitment transaction, the

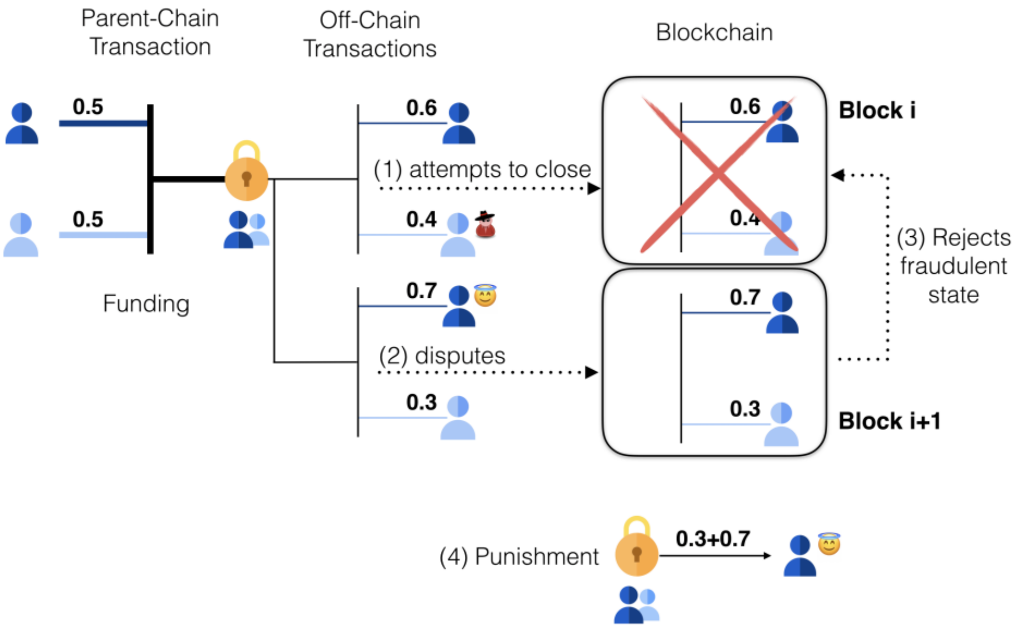


Fig. 56. Lightning network channel closing. In this example, the channel is updated twice off-chain. A malicious channel party tries to close the channel with an outdated state favorable to them by broadcasting the corresponding commitment transaction. The malicious party must wait λ blocks before their portion is spendable. The honest counterparty sees the malicious transaction confirmed on the blockchain before the timelock expires, and submits a breach remedy transaction to claim all the funds in the channel - including the coins of the malicious party. [528]

other party can punish them by taking all of the coins in the channel with their breach remedy transaction. An example of this is shown in Figure 56.

Payment channels would have limited utility if they only allowed transactions between direct channel partners. This would require consumers to open channels with every business they frequent. Luckily, the Lightning Network allows payments to flow across a network of open payment channels: if Alice has an open channel with Bob, and Bob has an open channel with Carol, then Alice can pay Carol using Bob as an untrusted intermediary. This process can be seen in Figure 57 and utilizes HTLCs, as described in Section 16.1. The final recipient chooses a preimage, and sends its hash to the original sender. The parties along the payment path set up HTLCs with each other using this as the hashlock, with timelocks getting shorter with each payment channel the path crosses. The preimage is then passed backwards along the path in order to allow each payment to complete.

This payment method utilizing HTLCs is susceptible to the *wormhole attack*, which allows colluding users to steal transaction fees from intermediaries [530]. If Alice and Carol colluded (or were, in fact, a single entity), steps 6 and 7 in Figure 57 could be skipped, and

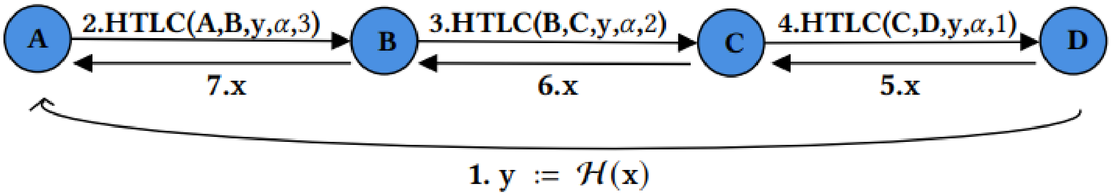


Fig. 57. Lightning Network payment. The notation $HTLC(Alice, Bob, y, \alpha, t)$ means that: (1) If timeout t expires, Alice can get back the α coins she locked, and (2) If Bob reveals a value x such that $H(x) = y$ before timeout t, then Alice will pay α coins to Bob. [529]

Carol could directly share the secret preimage with Alice instead of Bob. In this case, Bob's timelock will be left to expire, but Alice can complete the payment afterward because her timelock is longer. In the end, Bob will not collect his fee, leaving Alice and Carol with extra funds. This vulnerability stems from the use of the same preimage across the entire payment but can be solved using a cryptographic primitive called adaptor signatures. An adaptor signature is essentially a promise that the publication of an agreed-upon signature will reveal a secret value.

While payment channels are useful for payments, more general use cases may require the use of state channels, which are like payment channels but for arbitrary smart contracts. In many cases, these take advantage of Turing-complete state machines, such as the EVM. However, Bitcoin-compatible state channels have also been proposed [531]. These use adaptor signatures to create generalized state channels capable of executing any operation that the underlying network (e.g., Bitcoin) supports but off-chain. Turing-completeness is also sometimes required for certain types of payment channels, such as Perun virtual channels, where once set up, intermediaries are no longer required to be online for payments between endpoints [532]. Virtual channels were later extended to the UTXO model using only signatures and timelocks in order to be Bitcoin-compatible [533, 534]. Perun was extended to a multi-hop environment and for generalized state channels in [535] (which introduced state channels contemporaneously with [536]). These virtual state channels were further extended to allow contracts with more than two parties in [537].

General state channels operate very similarly to payment channels. Alice and Bob can execute smart contract code off-chain by deploying a state channel, G. First, both parties agree to an initial state G_0 of G and then exchange signatures on the tuple $(G_0, 0)$, where the second item is a sequence number. The smart contract is then executed over time by exchanging further tuples of an agreed-upon state and sequence number. Let (G_s, s) be the most recent mutually signed state and sequence number. If Alice wants to call a function f of the smart contract using input x, she will execute the function call locally using the current state G_s. Next, she sends Bob the signed tuple $(G_{s+1}, s+1)$ along with f and x. If Bob believes the computation was accurate, he replies with his signature over $(G_{s+1}, s+1)$. The state channel can be closed cooperatively using both parties' signatures over the most recent state. If one party is uncooperative, the contract will accept a new mutually signed state with a higher sequence number during a dispute period, allowing an honest user to ensure that the most recent state is committed on-chain.

An early state channel proposal is Sprites [538], which requires a form of globally shared state that Bitcoin does not provide. Sprites, when used for payments, implements HTLCs in a globally shared smart contract such that the expiration time and hash-lock can be enforced in the single smart contract instead of along each channel individually, so collateral need not be locked for as long along a payment path. If an LN payment is transferred across a path of L channels, and Δ is the amount of time before a transaction can be committed on chain, then collateral must be locked for $\Theta(L\Delta)$ time. Sprites reduces the collateral cost to $\Theta(L+\Delta)$ by using a smart contract (dubbed the PreimageManager) that logs statements along the lines of "the preimage R of hash $H = H(R)$ was recorded on the ledger before time $T_{expiration}$." Sprites contracts have a dispute handler that will query the PreimageManager contract to determine whether R was revealed on time, ensuring that disputed channels close in a consistent way using the single time $T_{expiration}$.

One of the more significant shortcomings of payment and state channels is that participants must periodically go online and check the underlying ledger to see if their channel counterparty has published an old state. If so, they need to initiate a dispute in order to prevent being defrauded. To reduce the impact of this requirement, the idea of *watchtowers* has been proposed [539–543]. Watchtowers act as an online monitor that can handle disputes on behalf of a channel party while the party is offline. Watchtower constructions have a variety of trade-offs: some scale poorly, may not be incentive-compatible, may harm privacy, or may require more complicated channel constructions to work properly. Another alternative is to design state channels that remain secure under asynchrony, such as Brick, which only allows unilateral channel closure after getting approval from an external committee [544]. Note that this section only scratched the surface of the issues involved in payment and state channels. Most known attacks and privacy issues have not been discussed here nor were the complexities of network topology and payment routing explored.

18.2.2. Plasma and Rollups

Other second layer technologies include Plasma and an idea called "rollups." Like sidechains (Section 16.3), these schemes involve running a separate blockchain system in parallel to a primary, parent ledger. Unlike sidechains, however, the security of transactions on these chains is derived from the underlying parent chain, and they are not required to run and secure their own consensus algorithm.

A Plasma chain is a blockchain whose security is anchored to its parent chain but where security is maintained by having users submit fraud proofs in order to resolve disputes (fraud proofs are discussed in Section 15.5). The entity or entities running the Plasma chain submit block headers – including a commitment to the state of the Plasma chain – to a parent chain smart contract (e.g., on Ethereum). If anyone detects invalid state transitions on the Plasma chain, they can dispute the block by submitting a fraud proof along with a bond to prevent misbehavior. If the fraud proof is valid, the Plasma chain is rolled back past the offending block, and the block creator is penalized.

As with payment and state channels, withdrawing funds from a Plasma chain requires a delay. In this case, the delay is to allow other parties to submit fraud proofs that challenge the blocks in which those withdrawals were processed. Plasma chains operate in the UTXO model, and an exiting party submits a bitmap of UTXOs to withdraw funds. Anyone else may submit a bonded fraud proof that challenges the original bitmap to prove that some of those funds have already been spent. Several Plasma variants have been proposed, often differentiating themselves based on their withdrawal mechanism [545–548]. For example, the original Plasma [545] and the simplified "Minimum Viable Plasma" proposal [546] require each party to individually submit an exit transaction, and they are prioritized based on how old the relevant UTXOs are. This prevents invalid withdrawal requests from being processed before valid ones that are less likely to come "out of nowhere." Alternatively, the "More Viable Plasma" proposal [547] prioritizes withdrawals based on the youngest input referenced in the exit transaction.

The constructions discussed so far have a variety of problems. Besides the user experience issue of needing to wait for a full challenge period before withdrawals are processed, Plasma users need to keep track of and verify the entire Plasma chain in order to detect malicious behavior to initiate an exit in the first place. Further, it may be the case that all users of a Plasma chain need to exit at once if the Plasma chain operator becomes unavailable and does not serve relevant data to users. This "mass exit" scenario requires the entire state of the Plasma chain to be dumped onto the parent ledger, which could cause significant congestion and prevent fraud proofs from being processed on time.

An alternative construction, Plasma Cash, resolves some of these issues [548]. Plasma Cash reduces users' data-checking requirements by using non-fungible tokens (NFTs) and sparse Merkle Trees. With this, users need only keep track of their own coins rather than the whole chain. The recipient of transactions on a Plasma Cash chain is responsible for checking that the coins being spent have a valid history on the chain based on proofs supplied by the spender. A limitation of Plasma Cash is that it only works with fixed denominations, requiring each NFT to be spent in full in any given transaction. Unfortunately, both Plasma Cash and the original (fungible) Plasma are unable to prevent an adversary from forcing honest users to take some involuntary, potentially expensive (in terms of transaction fees) on-chain action, be it a mass exit for fungible Plasma or non-constant sized exits for honest users on Plasma Cash [549].

Rollups are an alternative to Plasma that address the data availability problem (and, thus, mass exits) by taking all of the transactions that occur on the rollup chain and committing some of the transaction metadata to the parent chain. In the Ethereum context, this is done by taking this "rolled up" data and posting it in a transaction's *calldata* field, which is a read-only portion of Ethereum transactions that supply function call arguments. Calldata is vastly cheaper than typical blockchain storage, making this approach efficient. By ensuring that this transaction data is always available, user liveness requirements and data availability assumptions can be dispensed with, thus improving substantially over Plasma (and state channels). Because transaction data is available on-chain and consensus is maintained over that rollup data, any user can process the full rollup when desired and thus detect fraud or initiate withdrawals. Further, by resolving the data availability problem, assets need not be mapped to owners, which allows rollups to be general-purpose. Instead of just handling payments, the full EVM can be run inside of a rollup [550].

Two types of rollups have been proposed: optimistic rollups and zk-rollups. In an optimistic rollup, all blocks are simply assumed to be valid unless proven otherwise, and fraud proofs can be submitted to challenge an invalid block. This means that, as with Plasma, withdrawals require a delay such that challenges can be submitted and the rollup chain can be rolled back to deal with fraud. For zk-rollups, each rollup block comes with a SNARK that proves that the included transactions were properly executed. As such, zk-rollups allow withdrawals nearly instantly, but they create a burden on a chain operator, who must create an expensive proof. Rollups are likely to figure prominently in Ethereum's attempts to scale.

19. Incentives

In distributed ledger systems – especially permissionless environments – an important challenge is to have the incentives of the various participants aligned such that rational participants behave honestly. Incentives have been discussed throughout this document already, but this section will introduce some of the more nuanced aspects of how incentives interface with the security of the network.

19.1. Block Rewards: Subsidies and Transaction Fees

The most important and frequently discussed source of incentives in distributed ledger systems is the block reward that is earned when a block is produced. The block reward consists of one or both of a block subsidy (newly minted cryptocurrency units) and transaction fees, which clients pay in order to prioritize their transaction for inclusion in a block. Generally speaking, block rewards should be allocated proportionally to hash rate or stake in order to maintain desirable properties, such as being resistant to Sybil attacks and miner collusion [551, 552].

The block subsidy is a result of the monetary policy of the network and provides an intuitive method of distributing newly minted coins. In the Bitcoin network, for example, the block subsidy began as 50 bitcoin per block but is halved every four years until a total of 21 million bitcoin exist, which should occur by approximately year 2140. In contrast, Ethereum's initial supply of 72 million ether was distributed in a sale and placed in the genesis block, while the initial block subsidy was 5 ether per block (ignoring uncle blocks). This has been reduced several times via hard fork. Some proof of stake coins will distribute the entire supply in the genesis block and have no subsidy. Others, such as Monero, reduce the issuance up to a point and then have a *tail emission*, where a constant quantity of monero is minted per block. This is sometimes called a *disinflationary* monetary policy.

As of 2023, block subsidies dominate the block reward for most cryptocurrencies, and transaction fees make up a trivially small portion of the total rewards in networks other than Bitcoin and Ethereum. However, since block subsidies in most networks decrease over time, transaction fees are expected to become a more and more important component of the block reward. The implications of having transaction fees become the primary means of funding network security are worth exploring. If transactions are being broadcast at approximately the same pace that they are put into blocks, miners who order transactions based on the order they are seen will generate revenue substantially less than those who process transactions greedily based on fees [553]. As a result, if a miner is deciding between several conflicting transactions to include in their block, they are strongly incentivized to mine the transactions with the highest fee rate. This justifies certain policies, such as *replace-by-fee*, where clients can replace transactions in other nodes' mempools by bidding up their transaction fee enough to cover the additional bandwidth that the transaction will consume. This also suggests that the censorship resistance for transactions in most cryptocurrencies is primarily a result of the transaction fees provided because the fee is the opportunity cost to miners for censoring the transaction. A related issue is that miners may have little incentive to propagate high-fee transactions because they would prefer to claim the fees for themselves [554]. However, doing so increases block propagation time and thus increases the risk that the miner's block becomes stale.

One of the earliest insights into Bitcoin's fee economics was that if there is no maximum block size limit, transaction fees will fall toward zero, resulting in negligible security for the system [555]. This is generally true if there is an abundance of block space. If there is not a competitive market for this space, then fees will tend toward zero. Unless there is a persistent block subsidy (or sufficiently limited block size), the system may experience instability and low security. This is discussed in more detail in the next section.

19.1.1. The Mining Gap and (the Absence of a) Block Subsidy

The block subsidy provides a fairly consistent reward for miners, whereas transaction fees will vary from block to block based on the supply of and demand for block space. When the block subsidy dominates transaction fee income, the reward for any given block is nearly constant. On the other hand, as the subsidy becomes dominated by transaction fees, the variance in block rewards becomes significant. This variance may cause instability in consensus by encouraging miners to abandon the longest chain rule and perform adversarial mining strategies instead.

Carlsten et al. investigated the rational behavior of miners in a regime with no block subsidy, transaction fees accruing in the mempool at a constant rate, no latency in transaction

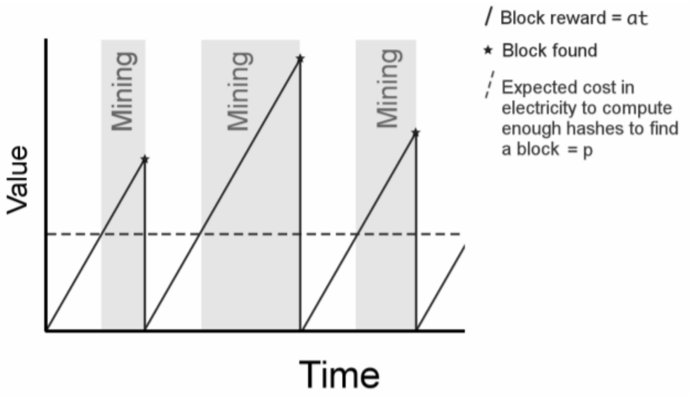


Fig. 58. Mining gap. The potential block reward is equal to the average transaction fee (a) multiplied by elapsed time (t) since the prior block was mined. [556]

or block propagation, and – crucially – miners having the space to include all available transactions in the next block if desired [556]. That is, the maximum allowed block size is significantly larger than what is in the mempool at any given time. In this scenario, it is possible for a *mining gap* to form: immediately after a block is mined, the mempool is empty and there are no transaction fees to put into the next block, so it makes sense for miners to shut down their machines until there is enough fee income to be worth the electricity and other operating costs. The idea behind the mining gap is illustrated in Figure 58.

This mining gap also incentivizes miners to *undercut* each other instead of moving the chain forward. If a block mined at height X claimed all available transaction fees in the mempool, then it makes little sense for competing miners to build off of that block at height $X+1$. Instead, they can gain an advantage by mining a different block at height X that only claims a portion of the total transaction fees. This leaves revenue available for other miners to be incentivized to mine at height $X+1$ on top of the adversary's block. Undercutting becomes an even more significant problem when nodes have heterogeneous internet connection speeds [237]. Miners may undercut each other more and more aggressively, forking with smaller and smaller fees and leaving more and more remaining available for others to mine on top of. This aggressive undercutting can lead to transactions failing to become confirmed as the chain fails to advance forward. An example of undercutting is shown in Figure 59.

Further, selfish mining is more profitable in this regime. A selfish miner's block on average will tend to be larger and include more transactions and, thus, rewards than when block subsidies exist and most block rewards are similar. This is because when the selfish miner has a long lead, the blocks that they mine are disproportionately large since it tends to take longer for the adversary to mine a block than the rest of the network. During this time, more transaction fees accumulate, and the selfish miner can still include transactions that were included in competing blocks on the public chain.

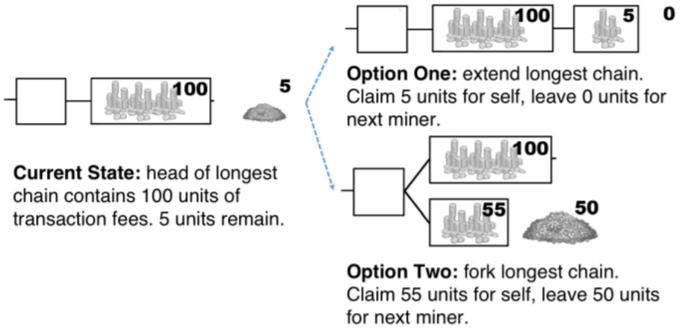


Fig. 59. Undercutting attack. [556]

The mining gap was further studied in [557], where the block subsidy was not reduced to zero but where transaction fees were nevertheless a significant component of the block reward. To this end, they define a metric – EBRR – that is the ratio between the expected base reward and the expected accumulated fees. Here, the base reward includes whatever transaction fees remain in the mempool after the most recent block has been mined. The expected accumulated fees is the expected block interval multiplied by the rate at which new transaction fees are added to the mempool. A mining gap is expected to form when the EBRR is approximately six but can be avoided if sufficiently high. An interesting result of that paper is that the consequences of the mining gap differ depending on the size of the miner. Because large miners are more likely to mine the next block and get a reward, they are willing to wait longer before turning their machines back on and cut electricity expenses while still having a good chance of winning. On the other hand, very small miners with low hash rate have almost no chance of winning unless they mine continuously and must therefore pay the electricity costs in order to maintain what little chance they have. In extreme scenarios, this could reduce mining utilization by up to 90%. There are several potential solutions to the issues related to the mining gap and undercutting attacks, each of which is intended to reduce the variance in block rewards:

1. The monetary policy in the network can include a block subsidy that is large enough to dominate transaction fees in perpetuity. This solution is simple but results in permanent inflation of the underlying asset.

2. The maximum block size can be sufficiently constrained such that, given the demand for block space, there is a nearly perpetual transaction backlog in the mempool. This option may increase transaction commit latency and fees for users. This was empirically shown to be effective in [558], where miners earn close to their fair share of rewards and avoid undercutting so long as they leave enough fees available in the mempool for the next miner.

3. The reward scheme can be designed such that the transaction fees for transactions in a given block are shared among miners of future blocks. Unfortunately, this has

several drawbacks that will be discussed in Section 19.3. Most significantly, it fails to solve the problem in the common case where out-of-band payments can be made to miners for transaction inclusion.

4. In some cases, the state machine may provide mechanisms that ensure that a transaction is only valid if included after particular blocks, such as the `lock_time` field in Bitcoin transactions. By default, Bitcoin wallets set the transaction `lock_time` to the height of the next block that they expect to be created. That is, if the wallet is aware of the chain tip being at block height X, the new transaction will only be considered valid in blocks of height $X+1$ or higher.

Finally, while the mining gap issue does not exist for proof-of-stake ledgers due to the lack of operational expenses like electricity, undercutting attacks are still a concern.

19.2. State Machines, Incentives, and Security

There are security ramifications to the interactions between the consensus algorithm and the state machine that is being replicated through it. The particulars of the state machine and the types of transactions that clients want executed can cause consensus instability through a variety of mechanisms. In this section, three issues will be discussed:

1. Miner extractable value
2. Mispriced computations
3. The Verifier’s Dilemma

Miner extractable value (MEV) is the total reward that a block producer can gain by manipulating the order of transactions in a particular time frame [559, 560]. Examples include front-running attacks (introduced in Section 7.1) and the undercutting attacks mentioned in the previous section. In many cases, MEV is captured by parties that are not block producers themselves, but miners are best positioned to capitalize on the existence of MEV given their privileged position in transaction ordering (they are essentially a "rushing adversary"). The ability to choose the order of transactions in the ledger is valuable in itself and can provide an additional source of revenue that exists exogenously to the consensus protocol. If MEV is significant enough, it may be used as a way of subsidizing chain reorg attacks. Empirical evidence suggests that MEV is a significant and growing phenomenon [559–563]. It has appeared in decentralized exchanges (DEXes), collectible games, gambling, name services, and initial coin offerings (ICOs) and was used by a prominent mining pool, F2Pool, to front-run the Status ICO [560]. While still a small minority, thousands of Ethereum blocks contain more MEV than honest block rewards [559, 563]. There are a variety of ways of exploiting MEV, and the most well-known are shown in Figure 60.

Front-running comes in two varieties: destructive and cooperative. When destructive front-running occurs, an attacker’s transaction is placed in front of a victim transaction in a

Fig. 60. Front-running and Miner Extractable Value. Attacker transactions are denoted T_A and marked in red, while the victim transaction is denoted T_V and is marked in blue. When destructive front-running occurs, the victim's transaction becomes invalid and fails to execute.

block, causing the victim transaction to become invalid. In cooperative front-running, the displaced victim transaction remains valid and executed but only after the attacker's transaction executes. Back-running is the opposite: the attacker wishes to have their transaction placed immediately after a particular target transaction. Finally, an attacker may try to clog the network with transactions in order to push a victim transaction into a future block. Contracts that have deadlines may create an incentive to clog the blockchain with transactions in order to suppress another user from having their transaction confirmed on time. For example, many of the layer 2 schemes discussed in Section 18.2 require parties to challenge fraudulent transactions within a given time limit.

These MEV building blocks may be profitable to exploit on their own, or they can be combined into more advanced attacks. For example, in a *sandwich attack*, a trader on a DEX will both front and back-run a victim transaction, T_V, simultaneously. The adversary listens on the network until a transaction arrives in their mempool that – after being executed – is expected to cause a change in price of an asset on the exchange. Say T_V is expected to increase the price of an asset. The attacker will first broadcast transaction TA_1, which buys the asset and uses it to cooperatively front-run T_V. This way, the asset is purchased before T_V raises the price. Next, the adversary broadcasts TA_2, which sells the asset and back-runs it immediately behind T_V.

In addition to DEXes, common financial applications built into smart contracts are lending and debt protocols. Typically, borrowers are required to over-collateralize their debt by locking, say, 150% of the value that the borrower wants to be loaned. For example, a user may lock $1,500 of ether in a smart contract that then provides them $1,000 worth of a stablecoin. If the collateral value decreases below $1,500 and the borrower fails to deposit more collateral, the original collateral is made available to liquidators to purchase at a discount in order to repay the debt. This can be exploited in multiple ways by liquidators. For instance, if the execution of a block creates a liquidation opportunity, the adversary can create transaction T_A that liquidates the collateral and keep bidding up its fee in order to destructively front-run competing liquidators. Alternatively, the adversary may spot a transaction T_V by an off-chain pricing oracle that creates a liquidation opportunity by adjusting the on-chain market price between the collateral asset and the borrowed asset. In this case, they can back-run T_V with their liquidation transaction. This has the advantage of avoiding a transaction fee bidding war.

Addressing MEV is challenging, especially in open blockchains where the ledger is public. Developers should be aware of MEV when designing applications and ideally design the application in such a way that transaction ordering is unimportant. Another possibility is to adjust the consensus layer so as to remove the ability of miners to arbitrarily order transactions. This is the approach taken in [564], which extends the Aequitas protocol mentioned in Section 7.1 to the permissionless setting. Finally, cryptography may be used to reduce an adversary's visibility of transactions in order to reduce the information available for the adversary to exploit. For example, HoneyBadgerBFT (Section 6.1) orders encrypted transactions, so an adversary would need to take advantage of meta-information in order to profitably front-run. Recently, a new cryptographic primitive called "multi-party timed commitments" has been proposed as a way for application developers to address front-running [565].

Another consensus security issue that arises due to the state machine is that the cost of executing certain operations may not accurately reflect the amount of computational effort or other resources required for the execution. Not every operation performed in the virtual machine has identical costs. Verifying a signature, for instance, is more computationally intensive than performing a simple addition, and the (gas) price of opcodes should reflect the costs of running them as accurately as possible. Finding accurate costs is a challenging task, particularly because these costs may not remain static over time or consistent across machines. The primary risk of mispriced state machine instructions is that they can enable network-wide denial-of-service attacks by creating transactions that are exceedingly difficult to verify.

The pricing of EVM opcodes has been studied empirically, and problems have been identified [566–568]. The main factor that seems to make gas prices improperly tuned is the role of state storage for smart contracts. The system state is too large to store in memory on most systems, so disk access is often required when a transaction must read or write to smart contract state. Not only is this a slow operation, but there can be high variability in its execution time depending on whether the data is cached already or if using an HDD or

SSD. This implies that high-end miners are advantaged relative to smaller hobbyist miners [566]. The mispricing opens up the possibility of malicious contracts that are extremely slow to verify. For example, [567] showed how to construct an Ethereum block that would take 93 seconds to verify – multiples longer than the expected block interval.

Opcode pricing was actually exploited on the live Ethereum network in September 2016, when an attacker constructed a malicious smart contract that required reading large amounts of state but where those operations were severely underpriced. This attack vector was fixed by increasing the gas cost of a variety of storage-accessing opcodes in EIP 150 [569]. A similar vulnerability existed on the live network but was fixed in April 2021 [570]. Bitcoin's more limited scripting support provides only marginal protection from this. It is possible to construct Bitcoin transactions that take an exceedingly long time to verify as well [571, 572]. One mitigation employed in Bitcoin is that developers have created standardized transaction templates for the most common types of transactions, and nodes have a policy where they refuse to propagate non-standard transactions. This means that in practice, the attacker would need to be a miner or collude with a miner.

A third security issue due to incentive alignment problems from state machines is called the *Verifier's Dilemma* [573]. When an honest node receives a newly mined block, it will verify that the block is valid before forwarding it to its peers and potentially mining on top of it himself. Unfortunately, this makes nodes susceptible to resource exhaustion attacks. For example, it is possible to design smart contracts that are slow to execute but that result in predictable state changes without the designer needing to execute it [574]. Fully verifying the block also imposes a delay for a miner, who would prefer to assume that the block is valid and immediately start working on the next block. That is, skipping verification can provide a revenue advantage for a miner by allowing them to find the next block more quickly. As a result, some miners may not validate blocks, which opens up the possibility of invalid blocks being accepted as part of the canonical consensus chain.

More broadly, the incentive to verify the correctness of transactions may not be sufficiently strong. In particular, as blocks grow larger and computational complexity increases, validators are more and more likely to skip verification to gain some advantage. If Ethereum were to raise the block gas limit significantly, non-verifying miners could gain substantially at the expense of those who verify [575]. Other research has shown that if validation takes 20% of the expected block time, then a non-validating miner with 33% of the hash rate can mine between 53% and 68% of main chain blocks, depending on network connectivity/delay [576]. Bitcoin's more limited scripting and the use of standardized transactions help mitigate this issue, but these are not complete solutions. In fact, on July 4, 2015, the Bitcoin network had a temporary six-block fork built on top of an invalid block, which suggests that a large segment of mining pools were not validating blocks that they received at the time [61].

In permissioned systems, the incentives will typically be exogenous to the system instead of denominated in a native token. The validator benefits because they can provide a better service and/or reduce the cost of providing the service. The costs for validators can be imposed legally instead of being in the form of work to prove or a capital investment in stake that can be slashed. However, this has not been tested legally in the real world, and complications such as jurisdictional questions and the possibility of accidentally producing invalid blocks (e.g., via a software bug) may make it non-trivial to impose those legal costs. If legal penalties cannot be relied upon, it can be rational for otherwise "honest" permissioned validators to accept invalid blocks under some circumstances, especially when blocks are very large or computationally intensive [577, 578].

Addressing the Verifier's Dilemma and the potential disincentive to validate blocks in general is challenging. Fundamentally, this requires that the cost of verifying blocks and transactions be substantially limited through having modest maximum block sizes or gas limits. Additionally, some of the alternative state machine techniques discussed in Sections 18.1.2 and 18.1.3 can limit the problem.

19.3. Alternative Transaction Fee Protocols

The current transaction fee mechanism used in Bitcoin and most existing cryptocurrency networks is a *multi-unit first-price auction* for block space. Clients attach a fee to their transactions, which is paid to the miner of the block that includes the transaction (in Bitcoin, the fee is determined implicitly as the difference between the input amounts and output amounts). Miners maximize their revenue by choosing the highest fee transactions (or more accurately, the highest fee *rate*) from the mempool up to the block size limit.

Ideally, clients would place bids that honestly represent the value that the user would get from transaction inclusion. Unfortunately, first-price auctions lead to strategic fee selection when bidding for block space. Instead of bidding the true value, the appropriate strategy for users is to deliberately underbid and then use techniques like replace-by-fee to increase their bid as needed. In practice, this can lead to rapid increases in fees as congestion increases and having those fees be "sticky" even when congestion goes back down. This makes both fees and the latency of transactions unpredictable to clients, which creates a negative user experience while consuming more bandwidth due to re-broadcasting the transactions. Another issue with the current scheme is that if there is insufficient demand for block space, clients can offer arbitrarily small fees above the marginal risk of the block becoming stale due to increased propagation delay, which creates a race to the bottom where miners may not be paid sufficiently to secure the network.

These issues led to several proposals for alternative fee mechanisms inspired by generalized *multi-unit second-price auctions* [579, 580]. In a second-price auction, the winning bidder only pays the bid of the second-highest bidder. In a generalized second-price auction (or K-th priced auction), K items are sold to the K-highest bidders, who each pay the $K+1$-th highest bid. This type of auction is known to lead to the dominant strategy where users' bids reflect their true value for the items. In the context of distributed ledgers, however, this type of auction is impossible to enforce because there is no consensus on the $K+1$-

th highest bid, which will only exist in miners' mempools. These two new fee protocols charge users the K-th bid, which is actually observable. In both cases, the advantage to be gained from strategic bidding goes toward zero as the number of clients issuing transactions increases.

While both new protocols operate similarly, they are optimized for different goals. The LSZ protocol from [579] is designed to decouple the fee market from the block size, such that an increase in the block size limit (or a decrease in demand for block space) will not lead to plummeting fees. That is, LSZ is intended to maximize fee revenue to improve security but comes at the expense of the social welfare of users. On the other hand, the BEOS protocol from [580] attempts to maximize social welfare and thus results in lower fees for users, as well as better throughput and latency for transactions than [579].

The LSZ protocol has clients specify in each transaction the maximal fee that user is willing to pay for transaction inclusion. Miners have full latitude to decide which transactions to put in blocks, but all transactions will end up paying the the same fee, which is equivalent to the lowest fee included in that block. For example, if the set of fees available in a miner's mempool is $\{5,2,1,1\}$, the miner will maximize their revenue by including only the transaction that pays five units of fees. Had the miner included all of those transactions in their block or just the first two, they would only end up claiming four units of fees.

The BEOS protocol differs from LSZ in two primary ways:

1. Under BEOS, miners are required to completely fill their blocks with transactions. If demand is insufficient, then the miner must stuff their block with "artificial" transactions, where the miner sends funds between addresses under their control.
2. The transaction fees are shared between miners over a period of B blocks. When a miner successfully mines a block, they are paid the average fee collected over the most recent B blocks, including their own.

Using the example above, the miner would be required to include all of the transactions available up to the block size limit and would thus generate four units of fees, which are then shared with the next B winning miners. This results in higher throughput and more stable fees for miners but lower overall mining rewards than either LSZ or the existing first-price auction mechanism. This may result in a lower amount of computational effort or stake used to secure the network. Reward sharing reduces the risk for miners to fork the ledger in order to try to double-spend but can improve censorship resistance by making it harder for miners to invalidate other miners' rewards. That said, it also weakens the censorship resistance benefit to users that comes with paying higher fees. Note that reward sharing is important in this scheme to reduce profits from miner manipulation of the fee mechanism. In the running example, assuming no reward sharing and a requirement of four transactions included in a block, a miner could gain by including three self-paying transactions with a fee of five units each. This would result in five units of fee revenue instead of four.

While these mechanisms both have desirable features, they have several problems. First, these schemes fail to account for the benefits miners can gain through MEV, as discussed in the previous section. Second, these mechanisms may be challenging to implement, particularly for UTXO ledgers. Because fees are typically implied by the difference between the inputs and the outputs, some mechanism is required in order to provide a refund for the surplus transaction fees included in their bid. This would require a substantial change to the system architecture and the way transactions are constructed. Third, and most importantly, these generalized second-price auction schemes do not have a mechanism to address out-of-band payments to miners or miner collusion with transaction senders. The BEOS scheme, in particular, creates an explicit incentive for side-dealing by sharing the reward. Miners are inclined to accept side payments from clients who want to transact in exchange for accepting transactions with low fees because miners cannot be forced to share the out-of-band payment. These bribes can be implemented easily in Bitcoin by including an extra output in a transaction that has a *scriptPubKey* set to `OP_TRUE` and is, thus, immediately spendable by anyone. Whoever mines the transaction that includes this output can then spend the output to themselves within the same block, accepting the bribe.

Not only can transactors bribe miners out-of-band, but miners can also collude with transactors to raise fees. If a transaction sender would have normally submitted a low fee of f_{low}, a miner can ask them to include a higher transaction fee of f_{high} instead and then pay the transactor $f_{high} - \frac{f_{low}}{2}$ [581]. This allows the miner to capture some of the revenue that would otherwise be lost by excluding transactions (in LSZ) or filling the block with dummy transactions (in BEOS, if rewards were not shared).

Both first- and second-price auctions leave much to be desired in the context of fee protocols. In addition to the issues already discussed above, an ideal fee mechanism would take into account the marginal social costs, or externalities, imposed on nodes for processing transactions. Transaction inclusion provides a private benefit to the sender but imposes computation, bandwidth, and storage costs on every node. To internalize these externalities, the fee mechanism ought to result in fees commensurate to the resources used. This can be done by either setting a quantity limit, like a maximum block size, or by fixing a minimum price. In traditional economic theory, if the marginal social cost of resource consumption is fixed but the marginal private benefit is decreasing, it is more efficient to set a price for the resource instead of setting quantity limits. However, when marginal social costs are increasing, quantity limits are superior to price setting [581]. An informal analysis of the marginal social costs of transactions suggests a decreasing marginal cost at low transaction throughput and rapidly increasing marginal costs with higher throughput (due to decreased security), which "suggests that a flat per-weight-unit in-protocol transaction fee, coupled with a hard limit at the point where the marginal social cost starts rapidly increasing, is superior to a pure weight limit-based regime" [581]. This reasoning ultimately led to the fee mechanism described in EIP-1559, which is currently deployed in several networks, including Ethereum and Filecoin [582].

EIP-1559 breaks the total transaction fee down into two components: an algorithmically computed *base fee* that is burned and a user-supplied "tip" that is collected by the miner of the transaction. The base fee does not depend on the transactions included in a block but rather is determined by the preceding blocks. A target block size, s_{target}, is selected at the start, and the maximum block size is double this amount. Whenever the size of the most recent block is greater than s_{target}, the base fee is adjusted upward; if it is smaller than the target, the base fee decreases. Specifically, if r_{pred} is the base fee of the preceding block and s_{pred} is the size of the preceding block, then the current block's base fee $r_{cur} := r_{pred} * (1 + \frac{1}{8} * \frac{s_{pred} - s_{target}}{s_{target}})$. Transactions specify a tip and a fee cap, and transactions will not be included in the chain unless the fee cap exceeds r_{cur}.

A game-theoretic analysis of the EIP-1559 mechanism was performed in [583]. Rational miners are disincentivized from including fake transactions in their blocks, and the mechanism provides no avenues for miners and users to profitably collude using side payments. Furthermore, the mechanism itself does not lead to a decrease in security against selfish mining or double-spending, except insofar as the fee burn reduces the total hash rate. As a result, the inflationary block subsidy takes on increased importance. In addition, the optimal bidding strategy for users – except during sudden demand spikes – is to set their transaction's fee cap to their true value of having the transaction included. A separate analysis of the dynamics of EIP-1559 found that with steady transaction demand, there can be chaotic periods with alternating full and empty blocks. This is far more likely to occur if user valuations for what to tip are similar to each other and, thus, have low variance [584]. Another variant of EIP-1559 that eliminates the first-price auction optimal bidding issue during demand spikes has been proposed as well [585].

To review, first-price auctions, as currently used in most permissionless ledger systems, make reasoning about fees challenging for users. The optimal bid for the user depends on bids offered by other users at the same time. Second-price auctions can be manipulated by miners by stuffing blocks with their own transactions instead of real user ones. However, these strategic issues do not arise in fixed-price sales, so establishing a base fee independent of transactions in the current block is a reasonable way of easing the burden of fee estimation on users. If the user values transaction inclusion more than the base fee, they bid the base fee plus enough to compensate the miner for the marginal costs of transaction inclusion. The base fee must be burned or otherwise withheld from the block's miner to prevent side dealing, and it must adjust dynamically as demand for block space changes. There must be agreement over the base fee, so a proxy for demand is used (e.g., in the case of EIP-1559, a variable block size). If the entirety of the fee was burned, then miners would have no incentive to include transactions in their blocks at all. The extra tip resolves this issue while also providing a way for users to signal how much they value transaction inclusion.

References

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

[2] Pease M, Shostak R, Lamport L (1980) Reaching agreement in the presence of faults. *Journal of the ACM (JACM)* 27(2):228–234.

[3] LAMPORT L, SHOSTAK R, PEASE M (1982) The byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401.

[4] Castro M, Liskov B, et al. (1999) Practical byzantine fault tolerance. *OSDI*, Vol. 99, pp 173–186.

[5] Yaga D, Mell P, Roby N, Scarfone K (2018) Blockchain technology overview (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 8202. https://doi.org/10.6028/NIST.IR.8202.

[6] Bracha G (1987) Asynchronous byzantine agreement protocols. *Information and Computation* 75(2):130–143.

[7] Lamport L (1984) Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6(2):254–280.

[8] Schneider FB (1990) Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22(4):299–319.

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

[10] Blum E, Katz J, Loss J (2020) Network-agnostic state machine replication. *arXiv preprint arXiv:200203437* .

[11] Garay J, Kiayias A (2020) Sok: A consensus taxonomy in the blockchain era. *Cryptographers' Track at the RSA Conference* (Springer), pp 284–318.

[12] Guerraoui R, Kuznetsov P, Monti M, Pavlovič M, Seredinschi DA (2019) The consensus number of a cryptocurrency. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pp 307–316.

[13] Sliwinski J, Wattenhofer R (2019) Abc: Asynchronous blockchain without consensus. *arXiv preprint arXiv:190910926* .

[14] Baudet M, Danezis G, Sonnino A (2020) Fastpay: High-performance byzantine fault tolerant settlement. *arXiv preprint arXiv:200311506* .

[15] Collins D, Guerraoui R, Komatovic J, Monti M, Xygkis A, Pavlovic M, Kuznetsov P, Pignolet YA, Seredinschi DA, Tonkikh A (2020) Online payments by merely broadcasting messages (extended version). *arXiv preprint arXiv:200413184* .

[16] Auvolat A, Frey D, Raynal M, Taïani F (2020) Money transfer made simple. *hal-02861511v2f* .

[17] Cohen S, Keidar I (2021) Tame the wild with byzantine linearizability: Reliable broadcast, snapshots, and asset transfer. *arXiv preprint arXiv:210210597* .

[18] Georghiades Y, Streit R, Garg V (2021) Who needs consensus? a distributed monetary system between rational agents via hearsay. *arXiv preprint arXiv:210407574*

.

[19] Kuznetsov P, Pignolet YA, Ponomarev P, Tonkikh A (2021) Permissionless and asynchronous asset transfer [technical report]. *arXiv preprint arXiv:210504966* .

[20] Aiyer AS, Alvisi L, Clement A, Dahlin M, Martin JP, Porth C (2005) Bar fault tolerance for cooperative services. *Proceedings of the twentieth ACM symposium on Operating systems principles*, pp 45–58.

[21] McMenamin C, Daza V, Pontecorvi M (2020) Achieving state machine replication without honesty assumptions. *arXiv preprint arXiv:201210146* .

[22] Fischer MJ, Lynch NA, Paterson MS (1985) Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32(2):374–382.

[23] Ben-Or M (1983) Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract),". *Proceedings of the 2nd ACM Annual Symposium on Principles of Distributed Computing, Montreal, Quebec*, pp 27–30.

[24] Rabin MO (1983) Randomized byzantine generals. *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)* (IEEE), pp 403–409.

[25] Bracha G, Toueg S (1985) Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32(4):824–840.

[26] Dwork C, Lynch N, Stockmeyer L (1988) Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35(2):288–323.

[27] Spiegelman A (2020) In search for a linear byzantine agreement. *arXiv preprint arXiv:200206993* .

[28] Pass R, Shi E (2017) The sleepy model of consensus. *International Conference on the Theory and Application of Cryptology and Information Security* (Springer), pp 380–409.

[29] Guo Y, Pass R, Shi E (2019) Synchronous, with a chance of partition tolerance. *Annual International Cryptology Conference* (Springer), pp 499–529.

[30] Kim J, Mehta V, Nayak K, Shrestha N (2021) Making synchronous bft protocols secure in the presence of mobile sluggish faults. *IACR Cryptology ePrint Archive* :603.

[31] Pass R, Shi E (2017) Rethinking large-scale consensus. *2017 IEEE 30th Computer Security Foundations Symposium (CSF)* (IEEE), pp 115–129.

[32] Wang Q, Yu J, Chen S, Xiang Y (2020) Sok: Diving into dag-based blockchain systems. *arXiv preprint arXiv:201206128* .

[33] Douceur JR (2002) The sybil attack. *International workshop on peer-to-peer systems* (Springer), pp 251–260.

[34] Bowman M, Das D, Mandal A, Montgomery H (2021) On elapsed time consensus protocols. *IACR Cryptol ePrint Arch* :086.

[35] Yandamuri S, Abraham I, Nayak K, Reiter MK (2021) Communication-efficient bft protocols using small trusted hardware to tolerate minority corruption. *IACR Cryptol ePrint Arch* :184.

[36] Deuber D, Döttling N, Magri B, Malavolta G, Thyagarajan SAK (2018) Minting mechanisms for blockchain-or-moving from cryptoassets to cryptocurrencies. *IACR*

Cryptol ePrint Arch .

[37] Neiheiser R, Matos M, Rodrigues L (2021) The quest for scaling bft consensus through tree-based vote aggregation. *arXiv preprint arXiv:210312112* .

[38] Wang L, Shen X, Li J, Shao J, Yang Y (2019) Cryptographic primitives in blockchains. *Journal of Network and Computer Applications* 127:43–58.

[39] Partala J, Nguyen TH, Pirttikangas S (2020) Non-interactive zero-knowledge for blockchain: A survey. *IEEE Access* 8:227945–227961.

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

[41] Boneh D, Lynn B, Shacham H (2001) Short signatures from the weil pairing. *International conference on the theory and application of cryptology and information security* (Springer), pp 514–532.

[42] Dryja T, Liu QC, Narula N (2020) A lower bound for byzantine agreement and consensus for adaptive adversaries using vdfs. *arXiv preprint arXiv:200401939* .

[43] Boneh D, Bonneau J, Bünz B, Fisch B (2018) Verifiable delay functions. *Annual international cryptology conference* (Springer), pp 757–788.

[44] Pietrzak K (2018) Simple verifiable delay functions. *10th innovations in theoretical computer science conference (itcs 2019)* (Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik).

[45] Wesolowski B (2019) Efficient verifiable delay functions. *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer), pp 379–407.

[46] (2015) Not bitcoin xt. Accessed on April 13, 2021. Available at https://github.com/xtbit/notbitcoinxt.

[47] Daian P (2017) On soft fork security. Accessed on April 4, 2021. Available at https://pdaian.com/blog/on-soft-fork-security/.

[48] Folkson M (2021) Should block height or mtp or a mixture of both be used in a soft fork activation mechanism?. Accessed on April 13, 2021. Available at https://bitcoin.stackexchange.com/questions/103854/should-block-height-or-mtp-or-a-mixture-of-both-be-used-in-a-soft-fork-activatio.

[49] Buterin V (2017) Hard forks, soft forks, defaults and coercion. Accessed on April 13, 2021. Available at https://vitalik.ca/general/2017/03/14/forks_and_markets.html.

[50] Wuille P (2015) Bitcoin core and hard forks. Accessed on April 13, 2021. Available at https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-July/009515.html.

[51] Srinivasan BS, Lee L (2017) Quantifying decentralization. Accessed on April 20, 2021. Available at https://news.earn.com/quantifying-decentralization-e39db233c28e.

[52] Sztork P (2015) Measuring decentralization. Accessed on April 20, 2021. Available at https://www.truthcoin.info/blog/measuring-decentralization/.

[53] Kwon Y, Liu J, Kim M, Song D, Kim Y (2019) Impossibility of full decentralization in permissionless blockchains. *Proceedings of the 1st ACM Conference on Advances*

in Financial Technologies, pp 110–123.

[54] Gervais A, Capkun S, Karame GO, Gruber D (2014) On the privacy provisions of bloom filters in lightweight bitcoin clients. *Proceedings of the 30th Annual Computer Security Applications Conference*, pp 326–335.

[55] Osuntokun O, Akselrod A, Posen J (2017) Client side block filtering. Accessed on April 22, 2021. Available at https://github.com/bitcoin/bips/blob/master/bip-0157.mediawiki.

[56] Osuntokun O, Akselrod A (2017) Compact block filters for light clients. Accessed on April 22, 2021. Available at https://github.com/bitcoin/bips/blob/master/bip-0158.mediawiki.

[57] Hearn M (2012) [bitcoin-development] roadmap to getting users onto spv clients. Accessed on April 27, 2021. Available at https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2012-December/002083.html.

[58] Research B (2019) Bitcoin's initial block download. Accessed on April 22, 2021. Available at https://blog.bitmex.com/bitcoins-initial-block-download/.

[59] Todd P (2015) [bitcoin-development] fwd: Block size increase requirements. Accessed on April 22, 2021. Available at https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-June/008407.html.

[60] Zhang R, Preneel B (2017) On the necessity of a prescribed block validity consensus: Analyzing bitcoin unlimited mining protocol. *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pp 108–119.

[61] Bitcoinorg (2015) Some miners generating invalid blocks. Available at https://bitcoin.org/en/alert/2015-07-04-spv-mining.

[62] Dryja T (2019) Utreexo: A dynamic hash-based accumulator optimized for the bitcoin utxo set. *IACR Cryptol ePrint Arch* :611.

[63] Bonneau J, Meckler I, Rao V, Shapiro E (2020) Mina: Decentralized cryptocurrency at scale. Available at https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf.

[64] karalabe (2015) eth/63 fast synchronization algorithm. Accessed on April 28, 2021. Available at https://github.com/ethereum/go-ethereum/pull/188.

[65] jamesob (2019) assumeutxo. Accessed on April 28, 2021. Available at https://github.com/bitcoin/bitcoin/issues/15605.

[66] Loruenser T, Rainer B, Wohner F (2021) Towards a performance model for byzantine fault tolerant (storage) services. *arXiv preprint arXiv:210104489* .

[67] Bessani A, Santos M, Felix J, Neves N, Correia M (2013) On the efficiency of durable state machine replication. *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pp 169–180.

[68] Malkhi D, Reiter M (1998) Byzantine quorum systems. *Distributed computing* 11(4):203–213.

[69] Wang Q, Yu J, Peng Z, Bui VC, Chen S, Ding Y, Xiang Y (2020) Security analysis on dbft protocol of neo. *International Conference on Financial Cryptography and Data Security* :20–31.

[70] Wang Q, Li R, Chen S, Xiang Y (2021) Formal security analysis on dbft protocol of neo. *arXiv preprint arXiv:210507459* .

[71] Bessani A, Sousa J, Alchieri EE (2014) State machine replication for the masses with bft-smart. *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (IEEE), pp 355–362.

[72] Saltini R, Hyland-Wood D (2019) Correctness analysis of ibft. *arXiv preprint arXiv:190107160* .

[73] Saltini R (2019) Ibft liveness analysis. *2019 IEEE International Conference on Blockchain (Blockchain)* (IEEE), pp 245–252.

[74] Saltini R, Hyland-Wood D (2019) Ibft 2.0: A safe and live variation of the ibft blockchain consensus protocol for eventually synchronous networks. *arXiv preprint arXiv:190910194* .

[75] Moniz H (2020) The istanbul bft consensus algorithm. *arXiv preprint arXiv:200203613* .

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

[77] Wang G (2021) Sok: Understanding bft consensus in the age of blockchains. *Cryptology ePrint Archive* :911.

[78] Abraham I, Gueta G, Malkhi D, Alvisi L, Kotla R, Martin JP (2017) Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:171201367* .

[79] Guerraoui R, Knežević N, Quéma V, Vukolić M (2010) The next 700 bft protocols. *Proceedings of the 5th European conference on Computer systems*, pp 363–376.

[80] Gunn LJ, Liu J, Vavala B, Asokan N (2019) Making speculative bft resilient with trusted monotonic counters. *2019 38th Symposium on Reliable Distributed Systems (SRDS)* (IEEE), pp 133–13309.

[81] Danezis G, Hrycyszyn D (2018) Blockmania: from block dags to consensus. *arXiv preprint arXiv:180901620* .

[82] Schett MA, Danezis G (2021) Embedding a deterministic bft protocol in a block dag. *arXiv preprint arXiv:210209594* .

[83] Shi E (2019) Streamlined blockchains: A simple and elegant approach (a tutorial and survey). *International Conference on the Theory and Application of Cryptology and Information Security* (Springer), pp 3–17.

[84] Chan BY, Shi E (2020) Streamlet: Textbook streamlined blockchains. *IACR Cryptol ePrint Arch* 2020:88.

[85] Pass R, Shi E (2018) Thunderella: Blockchains with optimistic instant confirmation. *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer), pp 3–33.

[86] Chan TH, Pass R, Shi E (2018) Pili: A simple, fast, and robust family of blockchain protocols (Cryptology ePrint Archive, Report 2018/980, 2018. https://eprint. iacr. org . . .),

[87] Chan THH, Pass R, Shi E (2018) Pala: A simple partially synchronous blockchain.

IACR Cryptol ePrint Arch 2018:981.

[88] Yin M, Malkhi D, Reiter MK, Gueta GG, Abraham I (2018) Hotstuff: Bft consensus in the lens of blockchain. *arXiv preprint arXiv:180305069* .

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

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

[91] Baudet M, Ching A, Chursin A, Danezis G, Garillot F, Li Z, Malkhi D, Naor O, Perelman D, Sonnino A (2019) State machine replication in the libra blockchain. *The Libra Assn, Tech Rep* .

[92] Momose A, Cruz JP (2019) Force-locking attack on sync hotstuff. *IACR Cryptol ePrint Arch* :1484.

[93] Abraham I, Nayak K, Ren L, Shrestha N (2020) On the optimality of optimistic responsiveness. *IACR Cryptol ePrint Arch* :458.

[94] Bhat A, Bandarupalli A, Bagchi S, Kate A, Reiter MK (2021) Apollo–optimistically linear and responsive smr. *IACR Cryptol ePrint Arch* :180.

[95] Miller A, Xia Y, Croman K, Shi E, Song D (2016) The honey badger of bft protocols. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp 31–42.

[96] Liu C, Duan S, Zhang H (2020) Epic: Efficient asynchronous bft with adaptive security. *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (IEEE), pp 437–451.

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

[98] Guo B, Lu Z, Tang Q, Xu J, Zhang Z (2020) Dumbo: Faster asynchronous bft protocols. *IACR Cryptology ePrint Archive* :841.

[99] Cachin C, Tessaro S (2005) Asynchronous verifiable information dispersal. *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)* (IEEE), pp 191–201.

[100] Mostéfaoui A, Moumen H, Raynal M (2015) Signature-free asynchronous binary byzantine consensus with $t < n/3$, $o(n^2)$ messages, and o(1) expected time. *Journal of the ACM (JACM)* 62(4):1–21.

[101] Duan S, Reiter MK, Zhang H (2018) Beat: Asynchronous bft made practical. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp 2028–2041.

[102] Lu Y, Lu Z, Tang Q (2021) Bolt-dumbo transformer: Asynchronous consensus as fast as pipelined bft. *arXiv preprint arXiv:210309425* .

[103] Baird L (2016) The swirlds hashgraph consensus algorithm: Fair, fast, byzantine fault tolerance. *Swirlds, Inc Technical Report SWIRLDS-TR-2016* 1.

[104] Kelkar M, Zhang F, Goldfeder S, Juels A (2020) Order-fairness for byzantine con-

sensus (IACR Cryptology ePrint Archive, 2020: 269),

[105] Gagol A, Leśniak D, Straszak D, Świetek M (2019) Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 214–228.

[106] Keidar I, Kokoris-Kogias E, Naor O, Spiegelman A (2021) All you need is dag. *arXiv preprint arXiv:210208325* .

[107] Kursawe K (2020) Wendy, the good little fairness widget. *arXiv preprint arXiv:200708303* .

[108] Asayag A, Cohen G, Grayevsky I, Leshkowitz M, Rottenstreich O, Tamari R, Yakira D (2018) A fair consensus protocol for transaction ordering. *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (IEEE), pp 55–65.

[109] Zhang Y, Setty S, Chen Q, Zhou L, Alvisi L (2020) Byzantine ordered consensus without byzantine oligarchy. *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp 633–649.

[110] Avarikioti Z, Heimbach L, Schmid R, Wattenhofer R (2020) Fnf-bft: Exploring performance limits of bft protocols. *arXiv preprint arXiv:200902235* .

[111] Lev-Ari K, Spiegelman A, Keidar I, Malkhi D (2019) Fairledger: A fair blockchain protocol for financial institutions. *arXiv preprint arXiv:190603819* .

[112] Kuo PC, Chung H, Chao TW, Cheng CM (2020) Fair byzantine agreements for blockchains. *IEEE Access* 8:70746–70761.

[113] Küsters R, Rausch D, Simon M (2020) Accountability in a permissioned blockchain: Formal analysis of hyperledger fabric (full version). *IACR Cryptology ePrint Archive* .

[114] Ranchal-Pedrosa A, Gramoli V (2021) Agreement in the presence of disagreeing rational players: The huntsman protocol. *arXiv preprint arXiv:210504357* .

[115] Civit P, Gilbert S, Gramoli V (2019) Polygraph: Accountable byzantine agreement. *IACR Cryptol ePrint Arch* :587.

[116] Crain T, Gramoli V, Larrea M, Raynal M (2018) Dbft: Efficient leaderless byzantine consensus and its application to blockchains. *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)* (IEEE), pp 1–8.

[117] Crain T, Natoli C, Gramoli V (2018) Evaluating the red belly blockchain. *arXiv preprint arXiv:181211747* .

[118] Ranchal-Pedrosa A, Gramoli V (2020) Blockchain is dead, long live blockchain! accountable state machine replication for longlasting blockchain. *arXiv preprint arXiv:200710541* .

[119] Sheng P, Wang G, Nayak K, Kannan S, Viswanath P (2020) Bft protocol forensics. *arXiv preprint arXiv:201006785* .

[120] Golan-Gueta G, Abraham I, Grossman S, Malkhi D, Pinkas B, Reiter MK, Seredinschi DA, Tamir O, Tomescu A (2019) Sbft: A scalable and decentralized trust infrastructure. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* :568–580.

[121] Jalalzai MM, Busch C, Richard GG (2019) Proteus: A scalable bft consen-

sus protocol for blockchains. *2019 IEEE International Conference on Blockchain (Blockchain)* (IEEE), pp 308–313.

[122] De Angelis S, Aniello L, Baldoni R, Lombardi F, Margheri A, Sassone V (2018) Pbft vs proof-of-authority: applying the cap theorem to permissioned blockchain. *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy, February 6th - to - 9th, 2018* 2058.

[123] Ekparinya P, Gramoli V, Jourjon G (2019) The attack of the clones against proof-of-authority. *arXiv preprint arXiv:190210244* .

[124] Natoli C, Gramoli V (2017) The balance attack or why forkable blockchains are ill-suited for consortium. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (IEEE), pp 579–590.

[125] Ekparinya P, Gramoli V, Jourjon G (2018) Double-spending risk quantification in private, consortium and public ethereum blockchains. *arXiv preprint arXiv:180505004* .

[126] Shi E (2019) Analysis of deterministic longest-chain protocols. *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)* (IEEE), pp 122–12213.

[127] Kiayias A, Russell A (2018) Ouroboros-bft: A simple byzantine fault tolerant consensus protocol. *IACR Cryptol ePrint Arch* :1049.

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

[129] Sheff I, Wang X, van Renesse R, Myers AC (2021) Heterogeneous paxos. *24th International Conference on Principles of Distributed Systems (OPODIS 2020)* (Schloss Dagstuhl-Leibniz-Zentrum für Informatik).

[130] Kane D, Fackler A, Gągol A, Straszak D, Zamfir V (2021) Highway: Efficient consensus with flexible finality. *arXiv preprint arXiv:210102159* .

[131] Xiang Z, Malkhi D, Nayak K, Ren L (2021) Strengthened fault tolerance in byzantine fault tolerant replication. *arXiv preprint arXiv:210103715* .

[132] Naor O, Baudet M, Malkhi D, Spiegelman A (2019) Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:190905204* .

[133] Naor O, Keidar I (2020) Expected linear round synchronization: The missing link for linear byzantine smr. *arXiv preprint arXiv:200207539* .

[134] Bravo M, Chockler G, Gotsman A (2020) Making byzantine consensus live (extended version). *arXiv preprint arXiv:200804167* .

[135] Spiegelman A, Rinberg A (2019) Ace: Abstract consensus encapsulation for liveness boosting of state machine replication. *arXiv preprint arXiv:191110486* .

[136] Gelashvili R, Kokoris-Kogias L, Spiegelman A, Xiang Z (2021) Be prepared when network goes bad: An asynchronous view-change protocol. *arXiv preprint arXiv:210303181* .

[137] Bessani A, Alchieri E, Sousa J, Oliveira A, Pedone F (2020) From byzantine replication to blockchain: Consensus is only the beginning. *arXiv preprint arXiv:200414527* .

[138] Aştefanoaei L, Chambart P, Del Pozzo A, Tate E, Tucci S, Zălinescu E (2020) Tenderbake–classical bft style consensus for public blockchains. *arXiv preprint arXiv:200111965* .

[139] Hao X, Yu L, Zhiqiang L, Zhen L, Dawu G (2018) Dynamic practical byzantine fault tolerance. *2018 IEEE Conference on Communications and Network Security (CNS)* (IEEE), pp 1–8.

[140] Kuznetsov P, Tonkikh A (2020) Asynchronous reconfiguration with byzantine failures. *arXiv preprint arXiv:200513499* .

[141] Vizier G, Gramoli V (2018) Comchain: Bridging the gap between public and consortium blockchains. *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (IEEE), pp 1469–1474.

[142] Freitas de Souza L, Kuznetsov P, Rieutord T, Tucci-Piergiovanni S (2021) Accountability and reconfiguration: Self-healing lattice agreement. *arXiv preprint arXiv:210504909* .

[143] Martin JP, Alvisi L (2004) A framework for dynamic byzantine storage. *International Conference on Dependable Systems and Networks, 2004* (IEEE), pp 325–334.

[144] Alchieri EA, Bessani AN, da Silva Fraga J, Greve F (2008) Byzantine consensus with unknown participants. *International Conference On Principles Of Distributed Systems* (Springer), pp 22–40.

[145] Alchieri EAP, Bessani A, Greve F, da Silva Fraga J (2016) Knowledge connectivity requirements for solving byzantine consensus with unknown participants. *IEEE Transactions on Dependable and Secure Computing* 15(2):246–259.

[146] Khanchandani P, Wattenhofer R (2021) Byzantine agreement with unknown participants and failures. *arXiv preprint arXiv:210210442* .

[147] Cachin C, Tackmann B (2019) Asymmetric distributed trust. *arXiv preprint arXiv:190609314* .

[148] Cachin C, Zanolini L (2020) Asymmetric byzantine consensus. *arXiv preprint arXiv:200508795* .

[149] Mazieres D (2015) The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation* .

[150] Lokhava M, Losa G, Mazières D, Hoare G, Barry N, Gafni E, Jove J, Malinowsky R, McCaleb J (2019) Fast and secure global payments with stellar. *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp 80–96.

[151] García-Pérez Á, Schett MA (2019) Deconstructing stellar consensus (extended version). *arXiv preprint arXiv:191105145* .

[152] Kim M, Kwon Y, Kim Y (2019) Is stellar as secure as you think? *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE), pp 377–385.

[153] Lachowski Ł (2019) Complexity of the quorum intersection property of the federated byzantine agreement system. *arXiv preprint arXiv:190206493* .

[154] Gaul A, Khoffi I, Liesen J, Stüber T (2019) Mathematical analysis and algorithms for federated byzantine agreement systems. *arXiv preprint arXiv:191201365* .

[155] Florian M, Henningsen S, Scheuermann B (2020) The sum of its parts: Analysis of federated byzantine agreement systems. *arXiv preprint arXiv:200208101* .

[156] Bracciali A, Grossi D, de Haan R (2019) Decentralization in open quorum systems. *arXiv preprint arXiv:191108182* .

[157] Schwartz D, Youngs N, Britto A, et al. (2014) The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper 58* .

[158] Chase B, MacBrough E (2018) Analysis of the xrp ledger consensus protocol. *arXiv preprint arXiv:180207242* .

[159] Amores-Sesar I, Cachin C, Mićić J (2020) Security analysis of ripple consensus. *arXiv preprint arXiv:201114816* .

[160] MacBrough E (2018) Cobalt: Bft governance in open networks. *arXiv preprint arXiv:180207240* .

[161] Dwork C, Naor M (1992) Pricing via processing or combatting junk mail. *Annual International Cryptology Conference* (Springer), pp 139–147.

[162] Jakobsson M, Juels A (1999) Proofs of work and bread pudding protocols. *Secure information networks* (Springer), pp 258–272.

[163] Back A, et al. (2002) Hashcash-a denial of service counter-measure .

[164] Gupta D, Saia J, Young M (2020) Resource burning for permissionless systems. *arXiv preprint arXiv:200604865* .

[165] Garay J, Kiayias A, Panagiotakos G (2020) Blockchains from non-idealized hash functions (Cryptology ePrint Archive 2019/315, https://eprint. iacr. org/2019/315),

[166] Hanke T (2016) Asicboost-a speedup for bitcoin mining. *arXiv preprint arXiv:160400575* .

[167] Recabarren R, Carbunar B (2017) Hardening stratum, the bitcoin pool mining protocol. *Proceedings on Privacy Enhancing Technologies* 2017(3):57–74.

[168] StopAndDecrypt (2019) Betterhash: Decentralizing bitcoin mining with new hashing protocols. Available at https://medium.com/hackernoon/betterhash-decentralizing-bitcoin-mining-with-new-hashing-protocols-291de178e3e0.

[169] Miller A, Kosba A, Katz J, Shi E (2015) Nonoutsourceable scratch-off puzzles to discourage bitcoin mining coalitions. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp 680–691.

[170] Chepurnoy A, Saxena A (2020) Bypassing non-outsourceable proof-of-work schemes using collateralized smart contracts. *IACR Cryptol ePrint Arch* 2020:44.

[171] tevador (2019) Randomx. Available at https://github.com/tevador/RandomX.

[172] Han R, Sui Z, Yu J, Liu J, Chen S (2019) Fact and fiction: Challenging the honest majority assumption of permissionless blockchains. *IACR Cryptol ePrint Arch* .

[173] Kwon Y, Kim H, Shin J, Kim Y (2019) Bitcoin vs. bitcoin cash: Coexistence or downfall of bitcoin cash? *2019 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 935–951.

[174] Bissias G, Böhme R, Thibodeau D, Levine BN (2020) Pricing security in proof-of-

work systems. *arXiv preprint arXiv:201203706* .

[175] Garratt R, van Oordt MR (2019) Why fixed costs matter for proof-of-work based cryptocurrencies. *Available at SSRN* .

[176] Mueller P (2020) Cryptocurrency mining: Asymmetric response to price movement. *Available at SSRN 3733026* .

[177] Arnosti N, Weinberg SM (2018) Bitcoin: A natural oligopoly. *arXiv preprint arXiv:181108572* .

[178] Romiti M, Judmayer A, Zamyatin A, Haslhofer B (2019) A deep dive into bitcoin mining pools: An empirical analysis of mining shares. *arXiv preprint arXiv:190505999* .

[179] Cong LW, He Z, Li J (2019) Decentralized mining in centralized pools. *The Review of Financial Studies* .

[180] Chatzigiannis P, Baldimtsi F, Griva I, Li J (2019) Diversification across mining pools: Optimal mining strategies under pow. *arXiv preprint arXiv:190504624* .

[181] Elmandjra Y, Hsue D (2020) Bitcoin mining the evolution of a multibillion dollar industry. Available at https://research.ark-invest.com/hubfs/1_Download_Files_ARK-Invest/White_Papers/ARKInvest_031220_Whitepaper_BitcoinMining.pdf.

[182] Garay J, Kiayias A, Leonardos N (2017) The bitcoin backbone protocol with chains of variable difficulty. *Annual International Cryptology Conference* (Springer), pp 291–323.

[183] Chan TH, Ephraim N, Marcedone A, Morgan A, Pass R, Shi E (2017) Blockchain with varying number of players.

[184] Garay JA, Kiayias A, Leonardos N (2020) Full analysis of nakamoto consensus in bounded-delay networks. *IACR Cryptol ePrint Arch* :277.

[185] Davidson M, Diamond T (2020) On the profitability of selfish mining against multiple difficulty adjustment algorithms. *IACR Cryptol ePrint Arch* :94.

[186] Werner SM, Ilie DI, Stewart I, Knottenbelt WJ (2020) Unstable throughput: When the difficulty algorithm breaks. *arXiv preprint arXiv:200603044* .

[187] Meshkov D, Chepurnoy A, Jansen M (2017) Short paper: Revisiting difficulty control for blockchain systems. *Data Privacy Management, Cryptocurrencies and Blockchain Technology* (Springer), pp 429–436.

[188] Goren G, Spiegelman A (2019) Mind the mining. *Proceedings of the 2019 ACM Conference on Economics and Computation*, pp 475–487.

[189] Fiat A, Karlin A, Koutsoupias E, Papadimitriou C (2019) Energy equilibria in proof-of-work mining. *Proceedings of the 2019 ACM Conference on Economics and Computation*, pp 489–502.

[190] Rosenfeld M (2011) Analysis of bitcoin pooled mining reward systems. *arXiv preprint arXiv:11124980* .

[191] Luu L, Saha R, Parameshwaran I, Saxena P, Hobor A (2015) On power splitting games in distributed computation: The case of bitcoin pooled mining. *2015 IEEE 28th Computer Security Foundations Symposium* (IEEE), pp 397–411.

[192] Eyal I (2015) The miner's dilemma. *2015 IEEE Symposium on Security and Privacy*

(IEEE), pp 89–103.

[193] wizkid057 (2014) Eligius: 0% fee btc, 105% pps nmc, no registration, cppsrb. Available at https://bitcointalk.org/?topic=441465.msg7282674.

[194] Kwon Y, Kim D, Son Y, Vasserman E, Kim Y (2017) Be selfish and avoid dilemmas: Fork after withholding (faw) attacks on bitcoin. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp 195–209.

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

[196] Roughgarden T, Shikhelman C (2021) Ignore the extra zeroes: Variance-optimal mining pools. *International Conference on Financial Cryptography and Data Security* (Springer), pp 233–249.

[197] Katz J, Lazos P, Marmolejo-Cossío FJ, Zhou X (2021) Rpplns: Pay-per-last-n-shares with a randomised twist. *arXiv preprint arXiv:210207681* .

[198] Zolotavkin Y, García J, Rudolph C (2017) Incentive compatibility of pay per last n shares in bitcoin mining pools. *International Conference on Decision and Game Theory for Security* (Springer), pp 21–39.

[199] Eyal I, Sirer EG (2014) Majority is not enough: Bitcoin mining is vulnerable. *International conference on financial cryptography and data security* (Springer), pp 436–454.

[200] Bahack L (2013) Theoretical bitcoin attacks with less than half of the computational power (draft). *arXiv preprint arXiv:13127013* .

[201] Kiayias A, Koutsoupias E, Kyropoulou M, Tselekounis Y (2016) Blockchain mining games. *Proceedings of the 2016 ACM Conference on Economics and Computation*, pp 365–382.

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

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

[204] Göbel J, Keeler HP, Krzesinski AE, Taylor PG (2016) Bitcoin blockchain dynamics: The selfish-mine strategy in the presence of propagation delay. *Performance Evaluation* 104:23–41.

[205] Gervais A, Karame GO, Wüst K, Glykantzis V, Ritzdorf H, Capkun S (2016) On the security and performance of proof of work blockchains. *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security* (ACM), pp 3–16.

[206] Bai Q, Zhou X, Wang X, Xu Y, Wang X, Kong Q (2019) A deep dive into blockchain selfish mining. *ICC 2019-2019 IEEE International Conference on Communications (ICC)* (IEEE), pp 1–6.

[207] Leelavimolsilp T, Tran-Thanh L, Stein S (2018) On the preliminary investigation of

selfish mining strategy with multiple selfish miners. *arXiv preprint arXiv:180202218* .

[208] Marmolejo-Cossío FJ, Brigham E, Sela B, Katz J (2019) Competing (semi-) selfish miners in bitcoin. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 89–109.

[209] Zhang S, Zhang K, Kemme B (2020) Analysing the benefit of selfish mining with multiple players. *2020 IEEE International Conference on Blockchain (Blockchain)* (IEEE), pp 36–44.

[210] Grunspan C, Pérez-Marco R (2019) Selfish mining in ethereum. *arXiv preprint arXiv:190413330* .

[211] Ritz F, Zugenmaier A (2018) The impact of uncle rewards on selfish mining in ethereum. *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE), pp 50–57.

[212] Feng C, Niu J (2019) Selfish mining in ethereum. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (IEEE), pp 1306–1316.

[213] Shibuya Y, Yamamoto G, Kojima F, Shi E, Matsuo S, Laszka A (2021) Selfish mining attacks exacerbated by elastic hash supply. *arXiv preprint arXiv:210308007* .

[214] Gramoli V (2020) From blockchain consensus back to byzantine consensus. *Future Generation Computer Systems* 107:760–769.

[215] Kiayias A, Panagiotakos G (2015) Speed-security tradeoffs in blockchain protocols. *IACR Cryptol ePrint Arch* :1019.

[216] Pass R, Seeman L, Shelat A (2017) Analysis of the blockchain protocol in asynchronous networks. *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer), pp 643–673.

[217] Wei P, Yuan Q, Zheng Y (2018) Security of the blockchain against long delay attack. *International Conference on the Theory and Application of Cryptology and Information Security* (Springer), pp 250–275.

[218] Badertscher C, Maurer U, Tschudi D, Zikas V (2017) Bitcoin as a transaction ledger: A composable treatment. *Annual International Cryptology Conference* (Springer), pp 324–356.

[219] Badertscher C, Garay J, Maurer U, Tschudi D, Zikas V (2018) But why does it work? a rational protocol design treatment of bitcoin. *Annual international conference on the theory and applications of cryptographic techniques* (Springer), pp 34–65.

[220] Cojocaru A, Garay JA, Kiayias A, Song F, Wallden P (2019) The bitcoin backbone protocol against quantum adversaries. *IACR Cryptol ePrint Arch* :1150.

[221] Ni P, Li H, Pan D (2020) Analysis of bitcoin backbone protocol in the non-flat model. *Science China Information Sciences* 63(3):1–14.

[222] Zhao J (2019) An analysis of blockchain consistency in asynchronous networks: Deriving a neat bound. *arXiv preprint arXiv:190906587* .

[223] Garay JA, Kiayias A, Panagiotakos G (2020) Consensus from signatures of work. *Cryptographers' Track at the RSA Conference* (Springer), pp 319–344.

[224] Duong T, Chepurnoy A, Zhou HS (2018) Multi-mode cryptocurrency systems. *Pro-*

ceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts, pp 35–46.

[225] Dembo A, Kannan S, Tas EN, Tse D, Viswanath P, Wang X, Zeitouni O (2020) Everything is a race and nakamoto always wins. *arXiv preprint arXiv:200510484* .

[226] Gazi P, Kiayias A, Russell A (2020) Tight consistency bounds for bitcoin. *IACR Cryptol ePrint Arch* .

[227] Avarikioti G, Käppeli L, Wang Y, Wattenhofer R (2019) Bitcoin security under temporary dishonest majority. *International Conference on Financial Cryptography and Data Security* (Springer), pp 466–483.

[228] Badertscher C, Gazi P, Kiayias A, Russell A, Zikas V (2020) Consensus redux: Distributed ledgers in the face of adversarial supremacy. *IACR Cryptol ePrint Arch* :1021.

[229] Ren L (2019) Analysis of nakamoto consensus. *IACR Cryptol ePrint Arch* :943.

[230] Kiffer L, Rajaraman R, Shelat A (2018) A better method to analyze blockchain consistency. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp 729–744.

[231] Cojocaru A, Garay J, Kiayias A, Song F, Wallden P (2020) Post-quantum security of the bitcoin backbone and quantum multi-solution bernoulli search. *arXiv preprint arXiv:201215254* .

[232] Li J, Guo D, Ren L (2020) Close latency–security trade-off for the nakamoto consensus. *arXiv preprint arXiv:201114051* .

[233] Sankagiri S, Gandlur S, Hajek B (2021) The longest-chain protocol under random delays. *arXiv preprint arXiv:210200973* .

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

[235] Silva P, Vavřička D, Barreto J, Matos M (2020) Impact of geo-distribution and mining pools on blockchains: A study of ethereum. *50th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* .

[236] Xiao Y, Zhang N, Lou W, Hou YT (2020) Modeling the impact of network connectivity on consensus security of proof-of-work blockchain. *arXiv preprint arXiv:200208912* .

[237] Jain A, Siddiqui S, Gujar S (2021) We might walk together, but i run faster: Network fairness and scalability in blockchains. *arXiv preprint arXiv:210204326* .

[238] Shahsavari Y, Zhang K, Talhi C (2020) A theoretical model for block propagation analysis in bitcoin network. *IEEE Transactions on Engineering Management* .

[239] Corallo M (2016) Compact block relay. Available at https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki.

[240] Otsuki K, Banno R, Shudo K (2020) Quantitatively analyzing relay networks in bitcoin. *2020 IEEE International Conference on Blockchain (Blockchain)* (IEEE), pp 214–220.

[241] Budish E (2018) The economic limits of bitcoin and the blockchain (National Bureau of Economic Research),

[242] Bonneau J, Felten EW, Goldfeder S, Kroll JA, Narayanan A (2016) Why buy when you can rent? bribery attacks on bitcoin consensus. *Citeseer* .

[243] Liao K, Katz J (2017) Incentivizing blockchain forks via whale transactions. *International Conference on Financial Cryptography and Data Security* (Springer), pp 264–279.

[244] Teutsch J, Jain S, Saxena P (2016) When cryptocurrencies mine their own business. *International Conference on Financial Cryptography and Data Security* (Springer), pp 499–514.

[245] McCorry P, Hicks A, Meiklejohn S (2018) Smart contracts for bribing miners. *International Conference on Financial Cryptography and Data Security* (Springer), pp 3–18.

[246] Winzer F, Herd B, Faust S (2019) Temporary censorship attacks in the presence of rational miners. *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE), pp 357–366.

[247] Judmayer A, Stifter N, Zamyatin A, Tsabary I, Eyal I, Gazi P, Meiklejohn S, Weippl ER (2019) Pay-to-win: Cheap, crowdfundable, cross-chain incentive manipulation attacks on cryptocurrencies. *IACR Cryptol ePrint Arch* :775.

[248] Moroz DJ, Aronoff DJ, Narula N, Parkes DC (2020) Double-spend counterattacks: Threat of retaliation in proof-of-work systems. *arXiv preprint arXiv:200210736* .

[249] Rosenfeld M (2014) Analysis of hashrate-based double spending. *arXiv preprint arXiv:14022009* .

[250] Grunspan C, Pérez-Marco R (2019) On profitability of nakamoto double spend. *arXiv preprint arXiv:191206412* .

[251] Sompolinsky Y, Zohar A (2016) Bitcoin's security model revisited. *arXiv preprint arXiv:160509193* .

[252] Finney H (2011) Best practice for fast transaction acceptance - how high is the risk?. Available at https://bitcointalk.org/index.php?topic=3441.msg48384#msg48384.

[253] vector76 (2011) Fake bitcoins?. Available at https://bitcointalk.org/index.php?topic=36788.msg463391#msg463391.

[254] Miller A (2013) Feather-forks: enforcing a blacklist with sub-50% hash power. Available at https://bitcointalk.org/index.php?topic=312668.0.

[255] Rizun PR (2016) Subchains: A technique to scale bitcoin and improve the user experience. *Ledger* 1:38–52.

[256] Zamyatin A, Stifter N, Schindler P, Weippl ER, Knottenbelt WJ (2018) Flux: Revisiting near blocks for proof-of-work blockchains. *IACR Cryptol ePrint Arch* :415.

[257] Eyal I, Gencer AE, Sirer EG, Van Renesse R (2016) Bitcoin-ng: A scalable blockchain protocol. *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*, pp 45–59.

[258] Yin J, Wang C, Zhang Z, Liu J (2018) Revisiting the incentive mechanism of bitcoin-ng. *Australasian Conference on Information Security and Privacy* (Springer), pp 706–719.

[259] Niu J, Wang Z, Gai F, Feng C (2020) Incentive analysis of bitcoin-ng, revisited.

arXiv preprint arXiv:200105082 .

[260] Lerner SD (2015) Decor+ hop: A scalable blockchain protocol.

[261] Camacho P, Lerner SD (2016) Decor+ lami: A scalable blockchain protocol.

[262] Zhang R, Preneel B (2017) Publish or perish: A backward-compatible defense against selfish mining in bitcoin. *Cryptographers' Track at the RSA Conference* (Springer), pp 277–292.

[263] Zhang R, Preneel B (2019) Lay down the common metrics: Evaluating proof-of-work consensus protocols' security. *2019 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 175–192.

[264] Zhang R, Zhang D, Wang Q, Xie J, Preneel B (2020) Nc-max: Breaking the throughput limit of nakamoto consensus. *IACR Cryptol ePrint Arch* :1101.

[265] Zhang R, Zhang D, Wang Q, Wu S, Xie J, Preneel B (2022) Nc-max: Breaking the security-performance tradeoff in nakamoto consensus. *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022* (The Internet Society). Available at https://www.ndss-symposium.org/ndss-paper/auto-draft-255/.

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

[267] Kiayias A, Panagiotakos G (2017) On trees, chains and fast transactions in the blockchain. *International Conference on Cryptology and Information Security in Latin America* (Springer), pp 327–351.

[268] Lerner SD (2016) Uncle mining, an ethereum consensus protocol flaw. Available at https://bitslog.com/2016/04/28/uncle-mining-an-ethereum-consensus-protocol-flaw/.

[269] Chang SY, Park Y, Wuthier S, Chen CW (2019) Uncle-block attack: Blockchain mining threat beyond block withholding for rational and uncooperative miners. *International Conference on Applied Cryptography and Network Security* (Springer), pp 241–258.

[270] Werner SM, Pritz PJ, Zamyatin A, Knottenbelt WJ (2019) Uncle traps: harvesting rewards in a queue-based ethereum mining pool. *Proceedings of the 12th EAI International Conference on Performance Evaluation Methodologies and Tools*, pp 127–134.

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

[272] Fitzi M, Gazi P, Kiayias A, Russell A (2018) Parallel chains: Improving throughput and latency of blockchain protocols via parallel composition. *IACR Cryptol ePrint Arch* :1119.

[273] Fitzi M, Gaži P, Kiayias A, Russell A (2020) Ledger combiners for fast settlement. *IACR Cryptol ePrint Arch* :675.

[274] Wang X, Muppirala VV, Yang L, Kannan S, Viswanath P (2021) Securing parallel-chain protocols under variable mining power. *arXiv preprint arXiv:210502927* .

[275] Bagaria V, Kannan S, Tse D, Fanti G, Viswanath P (2019) Prism: Deconstructing the blockchain to approach physical limits. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp 585–602.

[276] Yang L, Bagaria V, Wang G, Alizadeh M, Tse D, Fanti G, Viswanath P (2019) Prism: Scaling bitcoin by 10,000 x. *arXiv preprint arXiv:190911261* .

[277] Li J, Guo D (2020) Continuous-time analysis of the bitcoin and prism backbone protocols. *arXiv preprint arXiv:200105644* .

[278] Li J, Guo D (2020) Liveness and consistency of bitcoin and prism blockchains: The non-lockstep synchronous case. *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (IEEE), pp 1–9.

[279] Wang G, Wang S, Bagaria V, Tse D, Viswanath P (2020) Prism removes consensus bottleneck for smart contracts. *arXiv preprint arXiv:200408776* .

[280] Birmpas G, Koutsoupias E, Lazos P, Marmolejo-Cossío FJ (2020) Fairness and efficiency in dag-based cryptocurrencies. *International Conference on Financial Cryptography and Data Security* (Springer), pp 79–96.

[281] Lewenberg Y, Sompolinsky Y, Zohar A (2015) Inclusive block chain protocols. *International Conference on Financial Cryptography and Data Security* (Springer), pp 528–547.

[282] Li C, Li P, Zhou D, Xu W, Long F, Yao A (2018) Scaling nakamoto consensus to thousands of transactions per second. *arXiv preprint arXiv:180503870* .

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

[284] Li C, Long F, Yang G (2020) Ghast: Breaking confirmation delay barrier in nakamoto consensus via adaptive weighted blocks. *arXiv preprint arXiv:200601072* .

[285] Sompolinsky Y, Lewenberg Y, Zohar A (2016) Spectre: A fast and scalable cryptocurrency protocol. *IACR Cryptol ePrint Arch* :1159.

[286] Sompolinsky Y, Zohar A (2018) Phantom: A scalable blockdag protocol. *IACR Cryptol ePrint Arch* :104.

[287] Sompolinsky Y, Zohar A (2020) Phantom and ghostdag: A scalable generalization of nakamoto consensus, .

[288] Popov S (2016) The tangle.

[289] Bramas Q (2018) The stability and the security of the tangle. *HAL archives-ouvertes hal-01716111* .

[290] Penzkofer A, Kusmierz B, Capossele A, Sanders W, Saa O (2020) Parasite chain detection in the iota protocol. *arXiv preprint arXiv:200413409* .

[291] Kusmierz B, Sanders W, Penzkofer A, Capossele A, Gal A (2019) Properties of the tangle for uniform random and random walk tip selection. *2019 IEEE International Conference on Blockchain (Blockchain)* (IEEE), pp 228–236.

[292] Cullen A, Ferraro P, King C, Shorten R (2019) Distributed ledger technology for iot: Parasite chain attacks. *arXiv preprint arXiv:190400996* .

[293] Bu G, Gürcan Ö, Potop-Butucaru M (2019) G-iota: Fair and confidence aware tangle. *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (IEEE), pp 644–649.

[294] Bu G, Hana W, Potop-Butucaru M (2019) Metamorphic iota. *arXiv preprint arXiv:190703628* .

[295] Bentov I, Hubácek P, Moran T, Nadler A (2017) Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies. *IACR Cryptol ePrint Arch* :300.

[296] Pass R, Shi E (2017) Hybrid consensus: Efficient consensus in the permissionless model. *31st International Symposium on Distributed Computing (DISC 2017)* (Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik).

[297] Abraham I, Malkhi D, Nayak K, Ren L, Spiegelman A (2016) Solida: A blockchain protocol based on reconfigurable byzantine consensus. *arXiv preprint arXiv:161202916* .

[298] Lewis-Pye A, Roughgarden T (2020) Resource pools and the cap theorem. *arXiv preprint arXiv:200610698* .

[299] King S, Nadal S (2012) Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper* .

[300] Vasin P (2014) Blackcoin's proof-of-stake protocol v2. Accessed on March 17, 2021. Available at https://blackcoin.co/blackcoin-pos-protocol-v2-whitepaper.pdf.

[301] Earls J (2017) The missing explanation of proof of stake version 3. Accessed on March 17, 2021. Available at http://earlz.net/view/2017/07/27/1904/the-missing-explanation-of-proof-of-stake-version.

[302] Nxt whitepaper. Accessed on March 17, 2021. Available at https://nxtdocs.jelurida.com/Nxt_Whitepaper.

[303] Delegated proof of stake (dpos). Accessed on March 18, 2021. Available at https://how.bitshares.works/en/master/technology/dpos.html.

[304] Günther S (2018) Lisk - the mafia blockchain. Accessed on March 18, 2021. Available at https://medium.com/coinmonks/lisk-the-mafia-blockchain-47248915ae2f.

[305] Li C, Palanisamy B (2019) Incentivized blockchain-based social media platforms: A case study of steemit. *Proceedings of the 10th ACM Conference on Web Science*, pp 145–154.

[306] Gersbach H, Mamageishvili A, Schneider M (2021) Vote delegation and misbehavior. *arXiv preprint arXiv:210208823* .

[307] Gersbach H, Mamageishvili A, Schneider M (2021) Vote delegation favors minority. *arXiv preprint arXiv:210208835* .

[308] Cevallos A, Stewart A (2020) A verifiably secure and proportional committee election rule. *arXiv preprint arXiv:200412990* .

[309] Saleh F (2020) Blockchain without waste: Proof-of-stake. *Available at SSRN 3183935* .

[310] Ganesh C, Orlandi C, Tschudi D, Zohar A (2020) Virtual asics: Generalized proof-of-stake mining in cryptocurrencies. *IACR Cryptol ePrint Arch* :791.

[311] Bentov I, Gabizon A, Mizrahi A (2016) Cryptocurrencies without proof of work. *International conference on financial cryptography and data security* (Springer), pp 142–157.

[312] Kanjalkar S, Kuo J, Li Y, Miller A (2019) Short paper: I can't believe it's not stake! resource exhaustion attacks on pos. *International Conference on Financial Cryptography and Data Security* (Springer), pp 62–69.

[313] Bentov I, Pass R, Shi E (2016) Snow white: Provably secure proofs of stake. *IACR Cryptol ePrint Arch* :919.

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

[315] Stütz R, Gaži P, Haslhofer B, Illum J (2020) Stake shift in major cryptocurrencies: An empirical study. *International Conference on Financial Cryptography and Data Security* (Springer), pp 97–113.

[316] Mills DL (2017) *Computer network time synchronization: the network time protocol on earth and in space* (CRC press).

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

[318] Deirmentzoglou E, Papakyriakopoulos G, Patsakis C (2019) A survey on long-range attacks for proof of stake protocols. *IEEE Access* 7:28712–28725.

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

[320] Deb S, Kannan S, Tse D (2020) Posat: Proof-of-work availability and unpredictability, without the work. *arXiv preprint arXiv:201008154* .

[321] Boneh D, Eskandarian S, Hanzlik L, Greco N (2020) Single secret leader election. *IACR Cryptol ePrint Arch* :25.

[322] Catalano D, Fiore D, Giunta E (2021) Efficient and universally composable single secret leader election from pairings. *IACR Cryptol ePrint Arch* :344.

[323] Kerber T, Kiayias A, Kohlweiss M, Zikas V (2019) Ouroboros crypsinous: Privacy-preserving proof-of-stake. *2019 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 157–174.

[324] Ganesh C, Orlandi C, Tschudi D (2019) Proof-of-stake protocols for privacy-aware blockchains. *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer), pp 690–719.

[325] Baldimtsi F, Madathil V, Scafuro A, Zhou L (2020) Anonymous lottery in the proof-of-stake setting. *IACR Cryptol ePrint Arch* :533.

[326] Kohlweiss M, Madathil V, Nayak K, Scafuro A (2021) On the anonymity guarantees of anonymous proof-of-stake protocols. *IACR Cryptol ePrint Arch* :409.

[327] Brown-Cohen J, Narayanan A, Psomas A, Weinberg SM (2019) Formal barriers to

longest-chain proof-of-stake protocols. *Proceedings of the 2019 ACM Conference on Economics and Computation*, pp 459–473.

[328] Bagaria V, Dembo A, Kannan S, Oh S, Tse D, Viswanath P, Wang X, Zeitouni O (2019) Proof-of-stake longest chain protocols: Security vs predictability. *arXiv preprint arXiv:191002218* .

[329] Neuder M, Moroz DJ, Rao R, Parkes DC (2019) Selfish behavior in the tezos proof-of-stake protocol. *arXiv preprint arXiv:191202954* .

[330] Neuder M, Moroz DJ, Rao R, Parkes DC (2020) Defending against malicious reorgs in tezos proof-of-stake. *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pp 46–58.

[331] Blum E, Kiayias A, Moore C, Quader S, Russell A (2020) The combinatorics of the longest-chain rule: Linear consistency for proof-of-stake blockchains. *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (SIAM), pp 1135–1154.

[332] Kiayias A, Quader S, Russell A (2020) Consistency of proof-of-stake blockchains with concurrent honest slot leaders. *arXiv preprint arXiv:200106403* .

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

[334] Rosu I, Saleh F (2020) Evolution of shares in a proof-of-stake cryptocurrency. *HEC Paris Research Paper No FIN-2019-1339* .

[335] Irresberger F (2018) Coin concentration of proof-of-stake blockchains. *Leeds University Business School Working Paper* .

[336] Chitra T (2019) Competitive equilibria between staking and on-chain lending. *arXiv preprint arXiv:200100919* .

[337] Chitra T, Evans A (2020) Why stake when you can borrow? *Available at SSRN 3629988* .

[338] Amoussou-Guenou Y, Del Pozzo A, Potop-Butucaru M, Tucci-Piergiovanni S (2019) On fairness in committee-based blockchains. *arXiv preprint arXiv:191009786* .

[339] Leonardos S, Reijsbergen D, Piliouras G (2019) Weighted voting on the blockchain: Improving consensus in proof of stake protocols. *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (IEEE), pp 376–384.

[340] (2021) Proof of stake crypto assets ranking and yields. Accessed on April 4, 2023. Available at https://www.stakingrewards.com/proof-of-stake.

[341] Karakostas D, Kiayias A, Larangeira M (2020) Account management in proof of stake ledgers. *IACR Cryptol ePrint Arch* :525.

[342] Brünjes L, Kiayias A, Koutsoupias E, Stouka AP (2018) Reward sharing schemes for stake pools. *arXiv preprint arXiv:180711218* .

[343] Karakostas D, Kiayias A, Nasikas C, Zindros D (2019) Cryptocurrency egalitarianism: A quantitative approach. *International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2019)* (Schloss Dagstuhl-Leibniz-Zentrum fuer

Informatik).

[344] Catalini C, Jagadeesan R, Kominers SD (2020) Markets for crypto tokens, and security under proof of stake. *Available at SSRN* .

[345] Badertscher C, Gaži P, Kiayias A, Russell A, Zikas V (2018) Ouroboros genesis: Composable proof-of-stake blockchains with dynamic availability. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp 913–930.

[346] Badertscher C, Gazi P, Kiayias A, Russell A, Zikas V (2019) Ouroboros chronos: Permissionless clock synchronization via proof-of-stake. *IACR Cryptol ePrint Arch* :838.

[347] Hanke T, Movahedi M, Williams D (2018) Dfinity technology overview series, consensus system. *arXiv preprint arXiv:180504548* .

[348] Abraham I, Malkhi D, Nayak K, Ren L (2018) Dfinity consensus, explored. *IACR Cryptol ePrint Arch* :1153.

[349] Ethereum 2.0 specifications. Available at https://github.com/ethereum/eth2.0-specs.

[350] Buterin V, Hernandez D, Kamphefner T, Pham K, Qiao Z, Ryan D, Sin J, Wang Y, Zhang YX (2020) Combining ghost and casper. *arXiv preprint arXiv:200303052* .

[351] Buterin V Vitalik's annotated ethereum 2.0 spec. Available at https://github.com/ethereum/annotated-spec/blob/master/phase0/beacon-chain.md.

[352] Drake J (2018) Ethereum 2.0 randomness using a verifiable delay function (vdf). Available at https://docs.google.com/presentation/d/1VKBMcEMaY6PQjDJK4yAFtxn0vD41eYXW05YA3vdTBsI/edit#slide=id.g4677b9389c_0_732.

[353] Azouvi S, McCorry P, Meiklejohn S (2018) Betting on blockchain consensus with fantomette. *arXiv preprint arXiv:180506786* .

[354] Rocket T, Yin M, Sekniqi K, van Renesse R, Sirer EG (2019) Scalable and probabilistic leaderless bft consensus through metastability. *arXiv preprint arXiv:190608936* .

[355] Auvolat A, Bromberg YD, Frey D, Taïani F (2021) Basalt: A rock-solid foundation for epidemic consensus algorithms in very large, very open networks. *arXiv preprint arXiv:210204063* .

[356] Popov S, Buchanan WJ (2019) Fpc-bi: Fast probabilistic consensus within byzantine infrastructures. *arXiv preprint arXiv:190510895* .

[357] Fitzi M, Gazi P, Kiayias A, Russell A (2020) Proof-of-stake blockchain protocols with near-optimal throughput. *IACR Cryptol ePrint Arch* :37.

[358] Buchman E, Kwon J, Milosevic Z (2018) The latest gossip on bft consensus. *arXiv preprint arXiv:180704938* .

[359] Amoussou-Guenou Y, Del Pozzo A, Potop-Butucaru M, Tucci-Piergiovanni S (2018) Correctness and fairness of tendermint-core blockchains. *arXiv preprint arXiv:180508429* .

[360] Amoussou-Guenou Y, Del Pozzo A, Potop-Butucaru M, Tucci-Piergiovanni S (2019) Dissecting tendermint. *International Conference on Networked Systems* (Springer), pp 166–182.

[361] Braithwaite S, Buchman E, Khoffi I, Konnov I, Milosevic Z, Ruetschi R, Widder J (2020) A tendermint light client. *arXiv preprint arXiv:201007031* .

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

[363] Goyal V, Li H, Raizes J (2021) Instant block confirmation in the sleepy model. *International Conference on Financial Cryptography and Data Security* .

[364] Dziembowski S, Faust S, Kolmogorov V, Pietrzak K (2015) Proofs of space. *Annual Cryptology Conference* (Springer), pp 585–605.

[365] Park S, Kwon A, Fuchsbauer G, Gaži P, Alwen J, Pietrzak K (2018) Spacemint: A cryptocurrency based on proofs of space. *International Conference on Financial Cryptography and Data Security* (Springer), pp 480–499.

[366] Cohen B, Pietrzak K (2019) The chia network blockchain.

[367] Bentov I, Lee C, Mizrahi A, Rosenfeld M (2014) Proof of activity: Extending bitcoin's proof of work via proof of stake. *ACM SIGMETRICS Performance Evaluation Review* 42(3):34–37.

[368] Karakostas D, Kiayias A (2020) Securing proof-of-work ledgers via checkpointing. *IACR Cryptol ePrint Arch* :173.

[369] Block A (2018) Chainlocks. Available at https://github.com/dashpay/dips/blob/master/dip-0008.md.

[370] Buterin V, Griffith V (2017) Casper the friendly finality gadget. *arXiv preprint arXiv:171009437* .

[371] Buterin V, Reijsbergen D, Leonardos S, Piliouras G (2019) Incentives in ethereum's hybrid casper protocol. *2019 IEEE international conference on blockchain and cryptocurrency (ICBC)* (IEEE), pp 236–244.

[372] Nrryuya (2019) Analysis of bouncing attack on ffg. Available at https://ethresear.ch/t/analysis-of-bouncing-attack-on-ffg/6113.

[373] Neu J, Tas EN, Tse D (2020) Ebb-and-flow protocols: A resolution of the availability-finality dilemma. *arXiv preprint arXiv:200904987* .

[374] Neu J, Tas EN, Tse D (2021) The availability-accountability dilemma and its resolution via accountability gadgets. *arXiv preprint arXiv:210506075* .

[375] Dinsdale-Young T, Magri B, Matt C, Nielsen JB, Tschudi D (2019) Afgjort: A partially synchronous finality layer for blockchains, .

[376] Stewart A, Kokoris-Kogia E (2020) Grandpa: a byzantine finality gadget. *arXiv preprint arXiv:200701560* .

[377] Azouvi S, Danezis G, Nikolaenko V (2019) Winkle: Foiling long-range attacks in proof-of-stake systems. *IACR Cryptol ePrint Arch* :1440.

[378] Neu J, Tas EN, Tse D (2020) Snap-and-chat protocols: System aspects. *arXiv preprint arXiv:201010447* .

[379] Sankagiri S, Wang X, Kannan S, Viswanath P (2020) The checkpointed longest chain: User-dependent adaptivity and finality. *arXiv preprint arXiv:201013711* .

[380] Luu L, Narayanan V, Zheng C, Baweja K, Gilbert S, Saxena P (2016) A secure

sharding protocol for open blockchains. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp 17–30.

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

[382] Zamani M, Movahedi M, Raykova M (2018) Rapidchain: Scaling blockchain via full sharding. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp 931–948.

[383] Wang J, Wang H (2019) Monoxide: Scale out blockchains with asynchronous consensus zones. *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp 95–112.

[384] Team Z, et al. (2017) The zilliqa technical whitepaper. *Retrieved September* 16:2019.

[385] Skidanov A, Polosukhin I (2019) Nightshade: Near protocol sharding design. *URL: https://nearprotocol com/downloads/Nightshade pdf* :39.

[386] Al-Bassam M, Sonnino A, Bano S, Hrycyszyn D, Danezis G (2017) Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:170803778* .

[387] Li S, Yu M, Avestimehr S, Kannan S, Viswanath P (2018) Polyshard: Coded sharding achieves linearly scaling efficiency and security simultaneously. *arXiv preprint arXiv:180910361* .

[388] Das S, Krishnan V, Ren L (2020) Efficient cross-shard transaction execution in sharded blockchains. *arXiv preprint arXiv:200714521* .

[389] Rana R, Kannan S, Tse D, Viswanath P (2020) Free2shard: Adaptive-adversary-resistant sharding via dynamic self allocation. 2005.09610.

[390] Wang C, Raviv N (2020) Low latency cross-shard transactions in coded blockchain. *arXiv preprint arXiv:201100087* .

[391] Hellings J, Hughes DP, Primero J, Sadoghi M (2020) Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing. *arXiv preprint arXiv:200804450* .

[392] Androulaki E, De Caro A, Elkhiyaoui K, Gorenflo C, Sorniotti A, Vukolic M (2020) Multi-shard private transactions for permissioned blockchains. *arXiv preprint arXiv:201008274* .

[393] David B, Magri B, Matt C, Nielsen JB, Tschudi D (2021) Gearbox: An efficient uc sharded ledger leveraging the safety-liveness dichotomy. *IACR Cryptol ePrint Arch* :211.

[394] Wang G, Shi ZJ, Nixon M, Han S (2019) Sok: Sharding on blockchain. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 41–61.

[395] Liu Y, Liu J, Salles MAV, Zhang Z, Li T, Hu B, Henglein F, Lu R (2021) Building blocks of sharding blockchain systems: Concepts, approaches, and open problems. *arXiv preprint arXiv:210213364* .

[396] Yu G, Wang X, Yu K, Ni W, Zhang JA, Liu RP (2020) Survey: Sharding in blockchains. *IEEE Access* 8:14155–14181.

[397] Avarikioti G, Kokoris-Kogias E, Wattenhofer R (2019) Divide and scale: Formalization of distributed ledger sharding protocols. *arXiv preprint arXiv:191010434* .

[398] Manshaei MH, Jadliwala M, Maiti A, Fooladgar M (2018) A game-theoretic analysis of shard-based permissionless blockchains. *IEEE Access* 6:78100–78112.

[399] Han R, Yu J, Zhang R (2020) Analysing and improving shard allocation protocols for sharded blockchains. *IACR Cryptol ePrint Arch* :943.

[400] Sonnino A, Bano S, Al-Bassam M, Danezis G (2019) Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. *arXiv preprint arXiv:190111218* .

[401] Mizrahi A, Rottenstreich O (2020) Blockchain state sharding with space-aware representations. *IEEE Transactions on Network and Service Management* .

[402] Buterin V (2018) A note on data availability and erasure coding. Available at https://github.com/ethereum/research/wiki/A-note-on-data-availability-and-erasure-coding.

[403] Al-Bassam M, Sonnino A, Buterin V (2018) Fraud and data availability proofs: Maximising light client security and scaling blockchains with dishonest majorities. *arXiv preprint arXiv:180909044* .

[404] Yu M, Sahraei S, Li S, Avestimehr S, Kannan S, Viswanath P (2020) Coded merkle tree: Solving data availability attacks in blockchains. *International Conference on Financial Cryptography and Data Security* (Springer), pp 114–134.

[405] Mitra D, Tauz L, Dolecek L (2020) Concentrated stopping set design for coded merkle tree: Improving security against data availability attacks in blockchain systems. *arXiv preprint arXiv:201007363* .

[406] Sheng P, Xue B, Kannan S, Viswanath P (2020) Aced: Scalable data availability oracle. *arXiv preprint arXiv:201100102* .

[407] Buterin V (2017) On sharding blockchains. *Sharding FAQ* .

[408] Buterin V (2017) The stateless client concept. Available at https://ethresear.ch/t/the-stateless-client-concept/172.

[409] .Kuszmaul, J.: Verkle trees. In: Verkle Trees, pp. 1–12 (2019). https://math.mit.edu/research/highschool/primes/materials/2018/Kuszmaul.pdf.

[410] Belchior R, Vasconcelos A, Guerreiro S, Correia M (2020) A survey on blockchain interoperability: Past, present, and future trends. *arXiv preprint arXiv:200514282* .

[411] Robinson P (2020) Consensus for crosschain communications. *arXiv preprint arXiv:200409494* .

[412] Zamyatin A, Al-Bassam M, Zindros D, Kokoris-Kogias E, Moreno-Sanchez P, Kiayias A, Knottenbelt WJ (2019) Sok: communication across distributed ledgers. *Cryptology ePrint Archive* :1128.

[413] Pagnia H, Gärtner FC (1999) On the impossibility of fair exchange without a trusted third party (Citeseer),

[414] Dziembowski S, Eckey L, Faust S (2018) Fairswap: How to fairly exchange digital goods. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp 967–984.

[415] Eckey L, Faust S, Schlosser B (2020) Optiswap: Fast optimistic fair exchange. *Proceedings of the 15th ACM Asia Conference on Computer and Communications Se-*

curity, pp 543–557.

[416] Shook JM, Simon S, Mell P (2019) A smart contract refereed data retrieval protocol with a provably low collateral requirement. *IACR Cryptol ePrint Arch* :541.

[417] Hall-Andersen M (2019) Fastswap: Concretely efficient contingent payments for complex predicates. *IACR Cryptol ePrint Arch* :1296.

[418] Janin S, Qin K, Mamageishvili A, Gervais A (2020) Filebounty: Fair data exchange. *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE), pp 357–366.

[419] TierNolan (2013) Alt chains and atomic transfers. Available at https://bitcointalk.org/index.php?topic=193281.msg2224949#msg2224949.

[420] Herlihy M (2018) Atomic cross-chain swaps. *Proceedings of the 2018 ACM symposium on principles of distributed computing*, pp 245–254.

[421] van der Meyden R (2019) On the specification and verification of atomic swap smart contracts. *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (IEEE), pp 176–179.

[422] Liu JA (2018) Atomic swaptions: cryptocurrency derivatives. *arXiv preprint arXiv:180708644* .

[423] Lesavre L, Varin P, Yaga D (2021) Blockchain networks: Token design and management overview (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 8301. https://doi.org/10.6028/NIST.IR.8301.

[424] Shlomovits O, Leiba O (2020) Jugglingswap: Scriptless atomic cross-chain swaps. *arXiv preprint arXiv:200714423* .

[425] Zakhary V, Agrawal D, Abbadi AE (2019) Atomic commitment across blockchains. *arXiv preprint arXiv:190502847* .

[426] Gugger J (2020) Bitcoin–monero cross-chain atomic swap. *IACR Cryptol ePrint Arch* :1126.

[427] Hoenisch P, Soriano del Pino L (2021) Atomic swaps between bitcoin and monero. *arXiv preprint arXiv:210112332* .

[428] Herlihy M, Liskov B, Shrira L (2019) Cross-chain deals and adversarial commerce. *arXiv preprint arXiv:190509743* .

[429] van Glabbeek R, Gramoli V, Tholoniat P (2019) Cross-chain payment protocols with success guarantees. *arXiv preprint arXiv:191204513* .

[430] Black M, Liu T, Cai T (2019) Atomic loans: Cryptocurrency debt instruments. *arXiv preprint arXiv:190105117* .

[431] Liu Z, Xiang Y, Shi J, Gao P, Wang H, Xiao X, Wen B, Hu YC (2019) Hyperservice: Interoperability and programmability across heterogeneous blockchains. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp 549–566.

[432] Fynn E, Bessani A, Pedone F (2020) Smart contracts on the move. *arXiv preprint arXiv:200405933* .

[433] Robinson P, Hyland-Wood D, Saltini R, Johnson S, Brainard J (2019) Atomic cross-

chain transactions for ethereum private sidechains. *arXiv preprint arXiv:190412079* .

[434] Robinson P, Ramesh R (2020) General purpose atomic crosschain transactions. *arXiv preprint arXiv:201112783* .

[435] Nissl M, Sallinger E, Schulte S, Borkowski M (2020) Towards cross-blockchain smart contracts. *arXiv preprint arXiv:201007352* .

[436] Ghaemi S, Rouhani S, Belchior R, Cruz RS, Khazaei H, Musilek P (2021) A pub-sub architecture to promote blockchain interoperability. *arXiv preprint arXiv:2101:12331* .

[437] Xu J, Ackerer D, Dubovitskaya A (2020) A game-theoretic analysis of cross-chain atomic swaps with htlcs. *arXiv preprint arXiv:201111325* .

[438] Han R, Lin H, Yu J (2019) On the optionality and fairness of atomic swaps. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 62–75.

[439] Rueegger J, Machado GS (2020) Rational exchange: Incentives in atomic cross chain swaps. *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (IEEE), pp 1–3.

[440] Xue Y, Herlihy M (2021) Hedging against sore loser attacks in cross-chain transactions. *arXiv preprint arXiv:210506322* .

[441] Zamyatin A (2016) *Merged Mining: Analysis of Effects and Implications*. Ph.D. thesis. PhD thesis. Master's thesis, Vienna University of Technology, 2016 (cit. on . . . , .

[442] Merged mining introduction. Available at https://tlu.tarilabs.com/merged-mining/merged-mining-scene/MergedMiningIntroduction.html.

[443] Research B (2020) The growth of bitcoin merge mining. Available at https://blog.bitmex.com/the-growth-of-bitcoin-merge-mining/.

[444] Judmayer A, Zamyatin A, Stifter N, Voyiatzis AG, Weippl E (2017) Merged mining: Curse or cure? *Data Privacy Management, Cryptocurrencies and Blockchain Technology* (Springer), pp 316–333.

[445] Back A, Corallo M, Dashjr L, Friedenbach M, Maxwell G, Miller A, Poelstra A, Timón J, Wuille P (2014) Enabling blockchain innovations with pegged sidechains .

[446] Sztork P, CryptAxe (2019) Bip 301. Available at https://github.com/bitcoin/bips/blob/master/bip-0301.mediawiki.

[447] Sztork P (2015) Drivechain - the simple two way peg. Available at http://www.truthcoin.info/blog/drivechain/.

[448] Sztork P, CryptAxe (2017) Drivechain documentation – hashrate escrows. Available at https://github.com/drivechain-project/docs/blob/master/bip1-hashrate-escrow.md.

[449] Weippl E (2018) Pitchforks in cryptocurrencies: Enforcing rule changes through offensive. *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2018 International Workshops, DPM 2018 and CBT 2018, Barcelona, Spain, September 6-7, 2018, Proceedings* (Springer), Vol. 11025, p 197.

[450] Karantias K, Kiayias A, Zindros D (2019) Proof-of-burn. *IACR Cryptol ePrint Arch*

2019:1096.

[451] Ali M, Blankstein A, Freedman MJ, Galabru L, Gupta D, Nelson J, Soslow J, Stanley P (2020) Pox: Proof of transfer mining with bitcoin. *Blockstack PBC* .

[452] Nelson J (2021) What kind of blockchain is stacks?. Available at https://stacks.org/stacks-blockchain.

[453] Robinson P (2018) Requirements for ethereum private sidechains. *arXiv preprint arXiv:180609834* .

[454] Dilley J, Poelstra A, Wilkins J, Piekarska M, Gorlick B, Friedenbach M (2016) Strong federations: An interoperable blockchain solution to centralized third-party risks. *arXiv preprint arXiv:161205491* .

[455] Nick J, Poelstra A, Sanders G (2020) Liquid: A bitcoin sidechain .

[456] Btc relay. Available at https://github.com/ethereum/btcrelay.

[457] Peace relay. Available at https://github.com/loiluu/peacerelay.

[458] Teutsch J, Straka M, Boneh D (2019) Retrofitting a two-way peg between blockchains. *arXiv preprint arXiv:190803999* .

[459] Zamyatin A, Harz D, Lind J, Panayiotou P, Gervais A, Knottenbelt W (2019) Xclaim: Trustless, interoperable, cryptocurrency-backed assets. *2019 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 193–210.

[460] Bünz B, Kiffer L, Luu L, Zamani M (2020) Flyclient: Super-light clients for cryptocurrencies. *2020 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 928–946.

[461] Garoffolo A, Kaidalov D, Oliynykov R (2020) Zendoo: a zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains. *arXiv preprint arXiv:200201847* .

[462] Westerkamp M, Eberhardt J (2020) zkrelay: Facilitating sidechains using zksnark-based chain-relays. *Contract* 1(2):3.

[463] Frauenthaler P, Sigwart M, Spanring C, Schulte S (2020) Testimonium: A cost-efficient blockchain relay. *arXiv preprint arXiv:200212837* .

[464] Burdges J, Cevallos A, Czaban P, Habermeier R, Hosseini S, Lama F, Alper HK, Luo X, Shirazi F, Stewart A, et al. (2020) Overview of polkadot and its design considerations. *arXiv preprint arXiv:200513456* .

[465] Goes C (2020) The interblockchain communication protocol: An overview. *arXiv preprint arXiv:200615918* .

[466] Koh J (2019) 5 differences between cosmos & polkadot. Available at https://medium.com/@juliankoh/5-differences-between-cosmos-polkadot-67f09535594b.

[467] Wang G (2021) Sok: Exploring blockchains interoperability. *IACR Cryptol ePrint Arch* :537.

[468] Kiayias A, Miller A, Zindros D (2017) Non-interactive proofs of proof-of-work. *IACR Cryptol ePrint Arch* 2017(963):1–42.

[469] Kiayias A, Zindros D (2019) Proof-of-work sidechains. *International Conference on Financial Cryptography and Data Security* (Springer), pp 21–34.

[470] Gaži P, Kiayias A, Zindros D (2019) Proof-of-stake sidechains. *2019 IEEE Sympo-*

sium on Security and Privacy (SP) (IEEE), pp 139–156.

[471] Neudecker T, Hartenstein H (2018) Network layer aspects of permissionless blockchains. *IEEE Communications Surveys & Tutorials* 21(1):838–857.

[472] Deshpande V, Badis H, George L (2018) Btcmap: Mapping bitcoin peer-to-peer network topology. *2018 IFIP/IEEE International Conference on Performance Evaluation and Modeling in Wired and Wireless Networks (PEMWN)* (IEEE), pp 1–6.

[473] Wang T, Zhao C, Yang Q, Zhang S (2020) Ethna: Analyzing the underlying peer-to-peer network of the ethereum blockchain. *arXiv preprint arXiv:201001373* .

[474] Daniel E, Rohrer E, Tschorsch F (2019) Map-z: Exposing the zcash network in times of transition. *2019 IEEE 44th Conference on Local Computer Networks (LCN)* (IEEE), pp 84–92.

[475] Cao T, Yu J, Decouchant J, Luo X, Verissimo P (2020) Exploring the monero peer-to-peer network. *Financial Cryptography and Data Security 2020, Sabah, 10-14 February 2020* .

[476] Delgado-Segura S, Bakshi S, Pérez-Solà C, Litton J, Pachulski A, Miller A, Bhattacharjee B (2019) Txprobe: Discovering bitcoin's network topology using orphan transactions. *International Conference on Financial Cryptography and Data Security* (Springer), pp 550–566.

[477] Miller A, Litton J, Pachulski A, Gupta N, Levin D, Spring N, Bhattacharjee B (2015) Discovering bitcoin's public topology and influential nodes. Available at http://www.cs.umd.edu/projects/coinscope/coinscope.pdf.

[478] Koshy P (2013) Coinseer: A telescope into bitcoin .

[479] Tramèr F, Boneh D, Paterson K (2020) Remote side-channel attacks on anonymous transactions. *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pp 2739–2756.

[480] Heilman E, Kendler A, Zohar A, Goldberg S (2015) Eclipse attacks on bitcoin's peer-to-peer network. *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp 129–144.

[481] Marcus Y, Heilman E, Goldberg S (2018) Low-resource eclipse attacks on ethereum's peer-to-peer network. *IACR Cryptol ePrint Arch* :236.

[482] Henningsen S, Teunis D, Florian M, Scheuermann B (2019) Eclipsing ethereum peers with false friends. *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE), pp 300–309.

[483] Apostolaki M, Zohar A, Vanbever L (2017) Hijacking bitcoin: Routing attacks on cryptocurrencies. *2017 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 375–392.

[484] Saad M, Cook V, Nguyen L, Thai MT, Mohaisen A (2019) Partitioning attacks on bitcoin: colliding space, time, and logic. *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)* (IEEE), pp 1175–1187.

[485] Saad M, Anwar A, Ravi S, Mohaisen D (2021) Hashsplit: Exploiting bitcoin asynchrony to violate common prefix and chain quality. *IACR Cryptol ePrint Arch* :299.

[486] Tran M, Choi I, Moon GJ, Vu AV, Kang MS (2020) A stealthier partitioning at-

tack against bitcoin peer-to-peer network. *IEEE Symposium on Security and Privacy (S&P)*.

[487] Paphitis A, Kourtellis N, Sirivianos M (2021) A first look into the structural properties and resilience of blockchain overlays. *arXiv preprint arXiv:210403044* .

[488] Apostolaki M, Marti G, Müller J, Vanbever L (2018) Sabre: Protecting bitcoin against routing attacks. *arXiv preprint arXiv:180806254* .

[489] Boverman A (2011) Timejacking & bitcoin. Available at http://culubas.blogspot.com/2011/05/timejacking-bitcoin_802.html.

[490] Company EC (2020) New releases: 2.1.1 and hotfix 2.1.1-1. Available at https://electriccoin.co/blog/new-releases-2-1-1-and-hotfix-2-1-1-1/.

[491] Imtiaz MA, Starobinski D, Trachtenberg A, Younis N (2019) Churn in the bitcoin network: Characterization and impact. *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)* (IEEE), pp 431–439.

[492] Kiffer L, Salman A, Levin D, Mislove A, Nita-Rotaru C (2021) Under the hood of the ethereum gossip protocol. *Proceedings of the Financial Cryptography and Data Security (FC'21) St George's, Grenada* .

[493] Motlagh SG, Mišić J, Mišić VB (2020) Impact of node churn in the bitcoin network. *IEEE Transactions on Network Science and Engineering* .

[494] Biryukov A, Pustogarov I (2015) Bitcoin over tor isn't a good idea. *2015 IEEE Symposium on Security and Privacy* (IEEE), pp 122–134.

[495] Blackshear S, Dill DL, Qadeer S, Barrett CW, Mitchell JC, Padon O, Zohar Y (2020) Resources: A safe language abstraction for money. *arXiv preprint arXiv:200405106* .

[496] Schwartz C (2019) Ethereum 2.0: A complete guide. ewasm.. Available at https://medium.com/chainsafe-systems/ethereum-2-0-a-complete-guide-ewasm-394cac756baf.

[497] Zheng S, Wang H, Wu L, Huang G, Liu X (2020) Vm matters: A comparison of wasm vms and evms in the performance of blockchain smart contracts. *arXiv preprint arXiv:201201032* .

[498] Hess T, Keefer R, Sirer EG Emin Sirer (2016) Ethereum's dao wars soft fork is a potential dos vector. Available at https://hackingdistributed.com/2016/06/28/ethereum-soft-fork-dos-vector/.

[499] Jedusor TE (2016) Mimblewimble. Available at https://scalingbitcoin.org/papers/mimblewimble.txt.

[500] Poelstra A (2016) Mimblewimble. Available at https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf.

[501] Fuchsbauer G, Orrù M, Seurin Y (2019) Aggregate cash systems: A cryptographic investigation of mimblewimble. *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer), pp 657–689.

[502] Reijsbergen D, Dinh TTA (2020) On exploiting transaction concurrency to speed up blockchains. *arXiv preprint arXiv:200306128* .

[503] Dickerson T, Gazzillo P, Herlihy M, Koskinen E (2019) Adding concurrency to smart

contracts. *Distributed Computing* :1–17.

[504] Saraph V, Herlihy M (2019) An empirical study of speculative concurrency in ethereum smart contracts. *arXiv preprint arXiv:190101376* .

[505] Bartoletti M, Galletta L, Murgia M (2020) A theory of transaction parallelism in blockchains. *arXiv preprint arXiv:201113837* .

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

[507] Anjana PS, Kumari S, Peri S, Rathor S, Somani A (2021) Optsmart: A space efficient optimistic concurrent execution of smart contracts. *arXiv preprint arXiv:210204875* .

[508] Paimani K (2021) Sonicchain: A wait-free, pseudo-static approach toward concurrency in blockchains. *arXiv preprint arXiv:210209073* .

[509] Sheff I, Wang X, Myers AC, van Renesse R (2018) A web of blocks. *arXiv preprint arXiv:180606978* .

[510] Yakovenko A (2020) Sealevel - parallel processing thousands of smart contracts. Available at https://medium.com/solana-labs/sealevel-parallel-processing-thousands-of-smart-contracts-d814b378192.

[511] Sasson EB, Chiesa A, Garman C, Green M, Miers I, Tromer E, Virza M (2014) Zerocash: Decentralized anonymous payments from bitcoin. *2014 IEEE Symposium on Security and Privacy* (IEEE), pp 459–474.

[512] Bünz B, Agrawal S, Zamani M, Boneh D (2020) Zether: Towards privacy in a smart contract world. *International Conference on Financial Cryptography and Data Security* (Springer), pp 423–443.

[513] Kosba A, Miller A, Shi E, Wen Z, Papamanthou C (2016) Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. *2016 IEEE symposium on security and privacy (SP)* (IEEE), pp 839–858.

[514] Bowe S, Chiesa A, Green M, Miers I, Mishra P, Wu H (2020) Zexe: Enabling decentralized private computation. *2020 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 947–964.

[515] Solomon R, Almashaqbeh G (2021) smartfhe: Privacy-preserving smart contracts from fully homomorphic encryption. *IACR Cryptol ePrint Arch* :133.

[516] Teutsch J, Reitwießner C (2019) A scalable verification solution for blockchains. *arXiv preprint arXiv:190804756* .

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

[518] Chacko JA, Mayer R, Jacobsen HA (2021) Why do my blockchain transactions fail? a study of hyperledger fabric (extended version). *arXiv preprint arXiv:210304681* .

[519] Guggenberger T, Sedlmeir J, Fridgen G, Luckow A (2021) An in-depth in-

vestigation of performance characteristics of hyperledger fabric. *arXiv preprint arXiv:210207731* .

[520] Kalodner H, Goldfeder S, Chen X, Weinberg SM, Felten EW (2018) Arbitrum: Scalable, private smart contracts. *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp 1353–1370.

[521] Wüst K, Matetic S, Egli S, Kostiainen K, Capkun S (2020) Ace: Asynchronous and concurrent execution of complex smart contracts. *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp 587–600.

[522] Wüst K, Diana L, Kostiainen K, Karame G, Matetic S, Capkun S (2019) Bitcontracts: Adding expressive smart contracts to legacy cryptocurrencies. *IACR Cryptol ePrint Arch* :857.

[523] Al-Bassam M (2019) Lazyledger: A distributed data availability ledger with client-side smart contracts. *arXiv preprint arXiv:190509274* .

[524] McCorry P, Möser M, Shahandasti SF, Hao F (2016) Towards bitcoin payment networks. *Australasian Conference on Information Security and Privacy* (Springer), pp 57–76.

[525] Decker C, Russell R, Osuntokun O (2018) eltoo: A simple layer2 protocol for bitcoin. *White paper: https://blockstream com/eltoo pdf* .

[526] Poon J, Dryja T (2016) The bitcoin lightning network: Scalable off-chain instant payments. Available at http://lightning.network/lightning-network-paper.pdf.

[527] Kiayias A, Litos OST (2020) A composable security treatment of the lightning network. *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)* (IEEE), pp 334–349.

[528] Gudgeon L, Moreno-Sanchez P, Roos S, McCorry P, Gervais A (2020) Sok: Layer-two blockchain protocols. *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24* (Springer), pp 201–226.

[529] Aumayr L, Abbaszadeh K, Maffei M (2022) Thora: Atomic and privacy-preserving multi-channel updates. *IACR Cryptol ePrint Arch* 2022:317.

[530] Malavolta G, Moreno-Sanchez P, Schneidewind C, Kate A, Maffei M (2019) Anonymous multi-hop locks for blockchain scalability and interoperability. *26th Annual Network and Distributed System Security Symposium, NDSS 2019*.

[531] Aumayr L, Ersoy O, Erwig A, Faust S, Hostakova K, Maffei M, Moreno-Sanchez P, Riahi S (2020) Generalized bitcoin-compatible channels. *IACR Cryptol ePrint Arch* :476.

[532] Dziembowski S, Eckey L, Faust S, Malinowski D (2019) Perun: Virtual payment hubs over cryptocurrencies. *2019 IEEE Symposium on Security and Privacy (SP)* (IEEE), pp 106–123.

[533] Aumayr L, Ersoy O, Erwig A, Faust S, Hostáková K, Maffei M, Moreno-Sanchez P, Riahi S (2020) Bitcoin-compatible virtual channels. *IACR Cryptol ePrint Arch* :554.

[534] Jourenko M, Larangeira M, Tanaka K (2020) Lightweight virtual payment channels. *International Conference on Cryptology and Network Security* (Springer), pp 365–

384.

[535] Dziembowski S, Faust S, Hostáková K (2018) General state channel networks. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp 949–966.

[536] Coleman J, Horne L, Xuanji L (2018) Counterfactual: Generalized state channels. Available at https://l4.ventures/papers/statechannels.pdf.

[537] Dziembowski S, Eckey L, Faust S, Hesse J, Hostáková K (2019) Multi-party virtual state channels. *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (Springer), pp 625–656.

[538] Miller A, Bentov I, Bakshi S, Kumaresan R, McCorry P (2019) Sprites and state channels: Payment networks that go faster than lightning. *International Conference on Financial Cryptography and Data Security* (Springer), pp 508–526.

[539] Avarikioti G, Laufenberg F, Sliwinski J, Wang Y, Wattenhofer R (2018) Towards secure and efficient payment channels. *arXiv preprint arXiv:181112740* .

[540] Khabbazian M, Nadahalli T, Wattenhofer R (2019) Outpost: A responsive lightweight watchtower. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 31–40.

[541] McCorry P, Bakshi S, Bentov I, Meiklejohn S, Miller A (2019) Pisa: Arbitration outsourcing for state channels. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 16–30.

[542] Avarikioti Z, Litos OST, Wattenhofer R (2020) Cerberus channels: Incentivizing watchtowers for bitcoin. *International Conference on Financial Cryptography and Data Security* (Springer), pp 346–366.

[543] Mirzaei A, Sakzad A, Yu J, Steinfeld R (2021) Fppw: A fair and privacy preserving watchtower for bitcoin. *IACR Cryptol ePrint Arch* :117.

[544] Avarikioti G, Kogias EK, Wattenhofer R (2019) Brick: Asynchronous state channels. *arXiv preprint arXiv:190511360* .

[545] Poon J, Buterin V (2017) Plasma: Scalable autonomous smart contracts. *White paper* :1–47.

[546] Buterin V (2018) Minimum viable plasma. Available at https://ethresear.ch/t/minimal-viable-plasma/426.

[547] Jones B, Fichter K (2018) More viable plasma. Available at https://ethresear.ch/t/more-viable-plasma/2160.

[548] Konstantopoulos G (2019) Plasma cash: Towards more efficient plasma constructions. *arXiv preprint arXiv:191112095* .

[549] Dziembowski S, Fabianski G, Faust S, Riahi S (2020) Lower bounds for off-chain protocols: Exploring the limits of plasma. *IACR Cryptol ePrint Arch* 2020:175.

[550] Buterin V (2021) An incomplete guide to rollups. Available at https://vitalik.ca/general/2021/01/05/rollup.html.

[551] Leshno J, Strack P (2019) Bitcoin: An impossibility theorem for proof-of-work based protocols. *Available at SSRN: https://ssrncom/abstract=3487355* .

[552] Chen X, Papadimitriou C, Roughgarden T (2019) An axiomatic approach to block

rewards. *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pp 124–131.

[553] Siddiqui S, Vanahalli G, Gujar S (2020) Bitcoinf: Achieving fairness for bitcoin in transaction fee only model. *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*, pp 2008–2010.

[554] Babaioff M, Dobzinski S, Oren S, Zohar A (2012) On bitcoin and red balloons. *Proceedings of the 13th ACM conference on electronic commerce*, pp 56–73.

[555] Houy N (2014) The economics of bitcoin transaction fees. *GATE WP* 1407.

[556] Carlsten M, Kalodner H, Weinberg SM, Narayanan A (2016) On the instability of bitcoin without the block reward. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp 154–167.

[557] Tsabary I, Eyal I (2018) The gap game. *Proceedings of the 2018 ACM SIGSAC conference on Computer and Communications Security*, pp 713–728.

[558] Gong T, Minaei M, Sun W, Kate A (2020) Undercutting bitcoin is not profitable. *arXiv preprint arXiv:200711480* .

[559] Daian P, Goldfeder S, Kell T, Li Y, Zhao X, Bentov I, Breidenbach L, Juels A (2019) Flash boys 2.0: Frontrunning, transaction reordering, and consensus instability in decentralized exchanges. *arXiv preprint arXiv:190405234* .

[560] Eskandari S, Moosavi S, Clark J (2019) Sok: Transparent dishonesty: front-running attacks on blockchain. *International Conference on Financial Cryptography and Data Security* (Springer), pp 170–189.

[561] Torres CF, Camino R, State R (2021) Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain. *arXiv preprint arXiv:210203347* .

[562] Qin K, Zhou L, Gervais A (2021) Quantifying blockchain extractable value: How dark is the forest? *arXiv preprint arXiv:210105511* .

[563] Zhou L, Qin K, Cully A, Livshits B, Gervais A (2021) On the just-in-time discovery of profit-generating transactions in defi protocols. *arXiv preprint arXiv:210302228* .

[564] Kelkar M, Deb S, Kannan S (2021) Order-fair consensus in the permissionless setting. *IACR Cryptol ePrint Arch* :139.

[565] Doweck Y, Eyal I (2020) Multi-party timed commitments. *arXiv preprint arXiv:200504883* .

[566] Yang R, Murray T, Rimba P, Parampalli U (2019) Empirically analyzing ethereum's gas mechanism. *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (IEEE), pp 310–319.

[567] Perez D, Livshits B (2019) Broken metre: Attacking resource metering in evm. *arXiv preprint arXiv:190907220* .

[568] Baird K, Jeong S, Kim Y, Burgstaller B, Scholz B (2019) The economics of smart contracts. *arXiv preprint arXiv:191011143* .

[569] Buterin V (2016) Eip 150. Available at https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md.

[570] Holst Swende M, Szilagyi P (2021) Dodging a bullet: Ethereum state problems.

Accessed on May 24, 2021. Available at https://blog.ethereum.org/2021/05/18/eth_state_problems/.

[571] Lerner SD (2013) New bitcoin vulnerability: A transaction that takes at least 3 minutes to be verified by a peer. Available at https://bitcointalk.org/?topic=140078.

[572] maaku (2015) alternatives-to-block-size-as-aggregate-resource-limits. Available at https://diyhpl.us/wiki/transcripts/scalingbitcoin/alternatives-to-block-size-as-aggregate-resource-limits/.

[573] Luu L, Teutsch J, Kulkarni R, Saxena P (2015) Demystifying incentives in the consensus computer. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp 706–719.

[574] Pontiveros BBF, Torres CF, et al. (2019) Sluggish mining: Profiting from the verifier's dilemma. *International Conference on Financial Cryptography and Data Security* (Springer), pp 67–81.

[575] Alharby M, Lunardi RC, Aldweesh A, van Moorsel A (2020) Data-driven model-based analysis of the ethereum verifier's dilemma. *arXiv preprint arXiv:200412768* .

[576] Das S, Awathare N, Ren L, Ribeiro VJ, Bellur U (2020) Better late than never; scaling computations in blockchain by delaying execution. *arXiv preprint arXiv:200511791* .

[577] Amoussou-Guenou Y, Biais B, Potop-Butucaru M, Tucci-Piergiovanni S (2019) Rationals vs byzantines in consensus-based blockchains. *arXiv preprint arXiv:190207895* .

[578] Amoussou-Guenou Y, Biais B, Potop-Butucaru M, Tucci-Piergiovanni S (2020) *Rational Behavior in Committee-Based Blockchains*. Ph.D. thesis. CEA List; LIP6, Sorbonne Université, CNRS, UMR 7606; HEC Paris, .

[579] Lavi R, Sattath O, Zohar A (2019) Redesigning bitcoin's fee market. *The World Wide Web Conference*, pp 2950–2956.

[580] Basu S, Easley D, O'Hara M, Sirer E (2019) Towards a functional fee market for cryptocurrencies. *Available at SSRN 3318327* .

[581] Buterin V (2018) Blockchain resource pricing. Available at https://ethresear.ch/t/draft-position-paper-on-resource-pricing/2838.

[582] Buterin V, Connor E, Dudley R, Slipper M, Norden I (2019) Fee market change for eth 1.0 chain. Available at https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1559.md.

[583] Roughgarden T (2020) Transaction fee mechanism design for the ethereum blockchain: An economic analysis of eip-1559. *arXiv preprint arXiv:201200854* .

[584] Leonardos S, Monnot B, Reijsbergen D, Skoulakis S, Piliouras G (2021) Dynamical analysis of the eip-1559 ethereum fee market. *arXiv preprint arXiv:210210567* .

[585] Ferreira MV, Moroz DJ, Parkes DC, Stern M (2021) Dynamic posted-price mechanisms for the blockchain transaction-fee market. *arXiv preprint arXiv:210314144* .