Beyond the Whitepaper: Where BFT Consensus Protocols Meet Reality

David Wong
zkSecurity

Denis Kolegov
Matter Labs

Ivan Mikushin
Σ0, zkSecurity

August 7, 2024

This paper presents a collection of lessons learned from analyzing the real-world security of various Byzantine Fault Tolerant (BFT) consensus protocol implementations. Drawing upon our experience as a team of security experts who have both developed and audited BFT systems, including BA★, HotStuff variants, Paxos variants, and DAG-based algorithms like Narwhal and Bullshark, we identify and analyze a variety of security vulnerabilities discovered in the translation of theoretical protocols into real-world code. Our analysis covers a range of issues, including subtle logic errors, concurrency bugs, cryptographic vulnerabilities, and mismatches between the theoretical model and the implementation. We provide detailed case studies illustrating these vulnerabilities, discuss their potential impact, and propose mitigation strategies. This work aims to provide valuable insights for both designers and implementers of BFT consensus protocols, ultimately contributing to the development of more secure and reliable distributed systems.

Keywords: BFT consensus, security analysis, implementation vulnerabilities, real-world security

1 Introduction

In 2008, Satoshi Nakamoto introduced Bitcoin: a novel way for anyone in the world to participate in the shared management of a database of account balances, as well as to participate in the policing of new transactions coming in. To contribute, anyone willing could play in a Bitcoin "lottery" by solving a computational puzzle (called "proof of work") and get a chance to win some newly printed Bitcoin (a process called "mining"). In addition, the lottery winner would also get the opportunity to write the next page of transactions into the ledger while at the same time pocketing transaction fees. In Bitcoin terms, that page is a block, and as they extend each other, they naturally form what we call a "blockchain". Note that Bitcoin's proof of work is often criticized as contributing to wasteful energy consumption across the world[1].

Bitcoin's lottery (more formally called "leader election") has many possible winners, leading to naturally occurring "forks" where multiple leaders are elected at the same time, leading to multiple proposals being made to extend the latest

[1] Bitcoin Energy Consumption Index compares the carbon footprint of Bitcoin to the one of Uzbekistan

block. As conflicting views of the state are an undesirable outcome, Bitcoin users have accepted a waiting period of about an hour called "confirmation time" to give time for the participants to agree on which branch they should continue building on top of. The more time they wait, the more confidence they get that they are looking at the canonical chain (which was originally defined as the longest one). The intuition behind this solution is that the proof of work mechanism makes it harder and harder for someone to compute a competing branch of the same size (or longer).

The "Waiting enough" solution is problematic for users, as transactions can be felt as slow to process (a metric called "finality"). In addition, it only really works against short-term forks, and as long as 51% of the computing power participating in the distributed protocol remains honest. A "51% attack" can happen when a majority of the participating computing power decides to collude to attack the network, leading to a fork that reorganizes a large portion of the ledger. This can, in turn, lead to thought-to-be-confirmed transactions being canceled or, worse, being sent to someone else (which we call a "double spend").

In addition, Bitcoin's solution to decentralize the security of a financial system and produce an ordering of transactions only works in the "synchronous" setting, in which the network is expected not to get partitioned for too long, and messages are expected to get delivered under a certain amount of time. In practice, this is a troublesome assumption, as networks tend to behave in unpredictable ways. Violation of this assumption can lead to two (or more) forks of Bitcoin coexisting for some dangerous period of time (until the network partition resolves).

On the other hand, long before Bitcoin was invented, in 1989, Leslie Lamport was submitting a paper called "The Part-Time Parliament", now famous for having introduced the Byzantine Fault Tolerance (BFT) problem. Not so long after, several solutions were proposed, including the seminal paper Practical Byzantine Fault Tolerance in 1999. It turns out that this now-old BFT problem was essentially the same problem that Bitcoin was trying to solve: how to tolerate faulty (or "Byzantine") nodes within a group of nodes that are trying to come to consensus on a value (or set of values).

Bitcoin did have a novel contribution to the literature of BFT consensus protocols: it introduced the idea of "permissionless" protocols, allowing anyone to join the group of consensus participants (which had been fixed, or "permissioned", in BFT papers so far). It did that by using the proof of work mechanism as a way to computationally limit the barrier of entry to participate in the protocol. (In the literature, this problem of preventing someone from creating infinite accounts is often referred to as "Sybil resistance".)

Once people realized that there were decades of untapped research that could benefit the blockchain community, many more BFT schemes were proposed and then adapted into real-world cryptocurrency systems (see Tendermint: Byzantine Fault Tolerance in the Age of Blockchains and Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing).

Today, most modern blockchain protocols have moved on from Bitcoin's consensus protocol (sometimes referred to as Nakamoto consensus) and have instead adopted protocols from the BFT literature. These protocols are orders of magnitude faster (both in terms of throughput and finality), have a minimal footprint, and can be proven not to fork in the harshest network conditions.

In our work as security researchers and auditors, we've had a front-row seat to the challenges of translating BFT theory into robust, real-world implementations. This paper distills the lessons we've learned from analyzing a wide array of BFT systems currently operating in production environments. To set the stage for this deep dive, we first survey the diverse landscape of BFT consensus protocols in use today in the next section. We then delve into the vulnerabilities and issues we've uncovered or seen – from subtle logic flaws to cryptographic oversights – providing practical insights that can guide the design and implementation of more secure and reliable distributed systems.

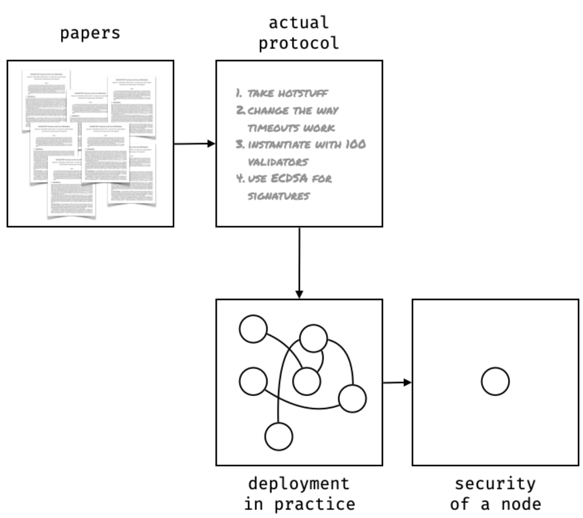


Figure 1: The different steps involved in productionizing a BFT consensus protocol.

2 The State of BFT Consensus Protocols in The Wild

Before delving into the meat of this paper, let's briefly take a look at the current state of BFT consensus protocols running in production and at the terminology that they use.

A BFT consensus protocol typically involves a fixed set of participants called validators (or nodes or replicas), who get elected as leaders to propose blocks of transactions (as we are mostly in a cryptocurrency context). Validators can then "vote" on proposals, so they can eventually get confirmed in the system, which we call "committed". Votes get aggregated into "quorum certificates", which can be used to prove that a quorum of participants has seen something. As there are many blocks to process in the system, a protocol is often iterated on through several consecutive views (which are often called rounds unless, confusingly, they themselves are made up of several rounds), with some views leading to commits and some views failing to commit anything.

The set of validators is fixed in a "genesis block" and then dynamically updated based on some application-level mechanism (for example, based on the amount of tokens that they have in the cryptocurrency). Different validator sets are split into different epochs.

We say that a validator is Byzantine or faulty when it acts maliciously: it does not follow the written protocol. This often translates into nodes "equivocating", meaning producing contradicting protocol messages. In addition, a validator who refuses to participate is also considered faulty, as honest validators will always attempt to participate even if it means that their message must be retried multiple times. The honesty of nodes is most often obtained through different incentivization mechanisms, which produce some "economic security"; this is out of the scope of this paper.

In BFT systems, a fork can never happen unless more than a certain threshold (often written as f) of validators decide to act Byzantine and collude at the same time. The property that a fork can never occur under these assumptions is called "safety", while the property that the protocol continually makes progress (i.e., that it doesn't get "stuck") is called "liveness". We will explain these in a second, but first, let's mention that BFT protocols are designed in one of these three models:

- **Synchronous model**: which is the model that Bitcoin uses, and which assumes that the network will never delay messages for more than some fixed upper bound of time (around an hour for Bitcoin).
- **Asynchronous model**: which assumes that network messages can be delayed ad-infinitum (but will eventually reach their destination).
- **Partially-synchronous model**: this is a mix between the two previous models, which assumes that while a network might behave asynchronously (for example, the network might partition), it will do so only for some definite-but-unknown (emphasis on unknown) period of time that will eventually be followed by a stable network that resembles a synchronous one.

As most BFT protocols have moved away from the synchronous model, as it is deemed an unrealistic assumption in practice, we will focus on partially-synchronous and asynchronous BFT consensus protocols in the rest of this paper. Between these two, the partially-synchronous model is often the most adopted one as it is quite pragmatic. On the other hand, asynchronous consensus protocols require additional primitives[2] which are annoying to implement, have higher latency than partially-synchronous consensus protocols, and are less understood, although they are not necessarily more complex, and have some benefits against some attacks (as we will see later).

In addition, most consensus protocols choose a model that tolerates around a third of Byzantine nodes. There exist different trade-offs, and we will mention some of the benefits of tolerating less faulty nodes in the section 3.1 ($f+1$ Attacks and Slashing).

In BFT whitepapers, the liveness of a consensus protocol is often deemed less important than its safety, and as such, their proofs and analysis are often much more handwavy. This is to some degree justified as in the popular partial-synchronous setting, for example, we pragmatically assume that periods of instability in the network won't last forever. Of course, although safety breaks lead to direct loss of funds, we recognize that network delays do also indirectly lead to loss of funds (e.g., a merchant can't get paid and so is not able to make a sale).

On the other hand, proving that a BFT protocol

[2] the 1985 FLP impossibility result is well-known for having proven that consensus is impossible in asynchronous settings without a common coin (Impossibility of Distributed Consensus with One Faulty Process). The intuition is that if the leader is known in advance, a network attacker could always force other messages to be prioritized, forcing participants to advance the protocol without the leader's messages. Choosing the leader randomly after committing to a set of messages fixes that.

is safe is often pretty straightforward, and papers often do so in small and self-contained proofs (see figure 2). In addition, ignoring the synchronous setting, the safety of consensus protocol is usually proven to hold in the harshest conditions: in periods of asynchrony (and thus in any lengthy partitioning of the network). That's right, even when the protocol is supposed to work in the partially-synchronous model (where a network could be partitioned for an unknown-but-bonded time), the protocol would still not fork in long periods of asynchrony.

The bugs examined in this study all affect the safety and/or liveness of the protocol, either directly or indirectly.

We refer to other papers for more information about these protocols, and from now on, assume that the reader knows about such BFT protocols. For example, see Reaching Consensus in the Byzantine Empire: A Comprehensive Review of BFT Consensus Algorithms, The Bedrock of Byzantine Fault Tolerance: A Unified Platform for BFT Protocols Analysis, Implementation, and Experimentation, and the great collaborative blog Decentralized Thoughts.

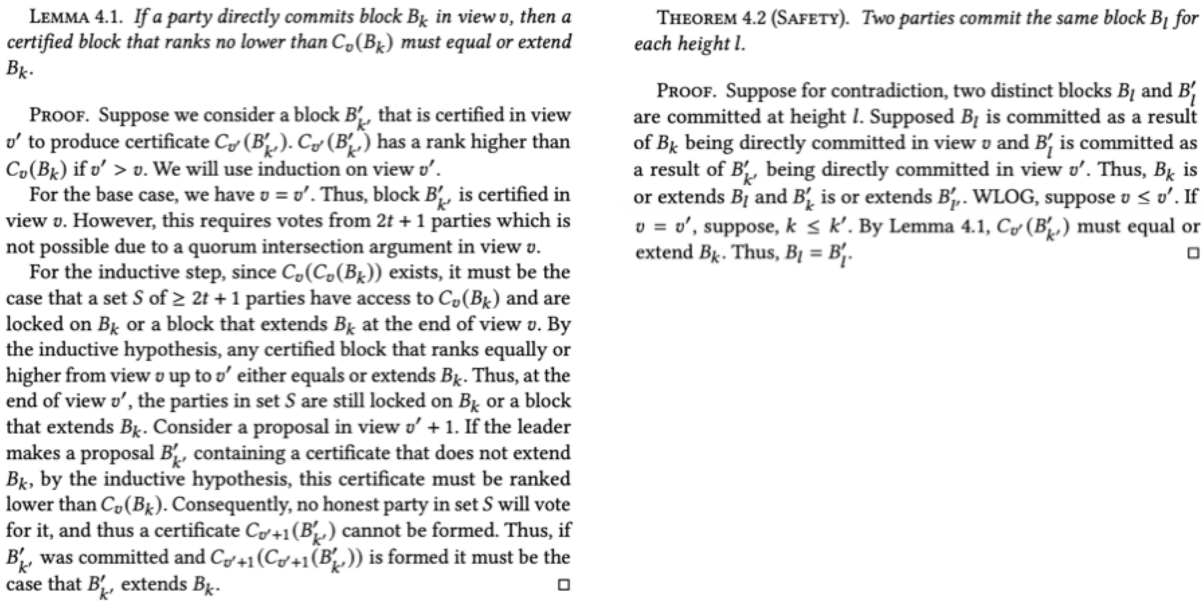
3 Left As An Exercise For The Reader

We start our tour of gotchas and vulnerabilities with a focus on what the BFT literature is missing. In the next examples, we'll look at bugs that stem from ambiguities or omissions in the original "paper protocols", which implementers are supposed to address by themselves. These gaps can be substantial and create fertile ground for numerous bugs, from the susceptibility to $f+1$ attacks despite slashing mechanisms, to the complexities of reconfiguration and long-range attacks arising from the impracticality of fixed validator sets. Assumptions about weighted voting, fast processing mechanisms, message retries, leader election, and node catch-up further expose the fragility of real-world systems. These challenges underscore the need for a deeper understanding of the intricacies involved in translating theoretical BFT protocols into robust and secure real-world systems.

3.1 $f+1$ Attacks and Slashing

In general, real-world instantiations of BFT protocols decide on an optimal number of participants n that they want to be able to support and then derive the number of Byzantine nodes f it can tolerate based on that (which is dictated by the BFT protocol that is implemented). If the number is too small, the process is restarted for a new increased value n. Rinse and repeat. For example, a lot of implementations use BFT protocols that work with $n = 3f+1$ participants, meaning that they can tolerate roughly a third of the participants acting Byzantine. Intuitively, the closer f is to the number of participants n, the more Byzantine participants the protocol can tolerate. (So a protocol that uses $n = 5f+1$ can tolerate less faulty nodes than a protocol that uses $n = 3f+1$.)

It was shown in several studies (see Good-case Latency of Byzantine Broadcast: A Complete Categorization and figure 4) that f out of $3f+1$ participants is the best ratio we can obtain outside of the synchronous model (which can tolerate half but is rarely used in practice due to the reasons we mentioned above). Yet, it imposes at least 3 "rounds" to reach an agreement, which is not ideal as each round adds more complexity



Lemma 4.1. *If a party directly commits block B_k in view v, then a certified block that ranks no lower than $C_v(B_k)$ must equal or extend B_k.*

Proof. Suppose we consider a block $B'_{k'}$ that is certified in view v' to produce certificate $C_{v'}(B'_{k'})$. $C_{v'}(B'_{k'})$ has a rank higher than $C_v(B_k)$ if $v' > v$. We will use induction on view v'.

For the base case, we have $v = v'$. Thus, block $B'_{k'}$ is certified in view v. However, this requires votes from $2t + 1$ parties which is not possible due to a quorum intersection argument in view v.

For the inductive step, since $C_v(C_v(B_k))$ exists, it must be the case that a set S of $\geq 2t + 1$ parties have access to $C_v(B_k)$ and are locked on B_k or a block that extends B_k at the end of view v. By the inductive hypothesis, any certified block that ranks equally or higher from view v up to v' either equals or extends B_k. Thus, at the end of view v', the parties in set S are still locked on B_k or a block that extends B_k. Consider a proposal in view $v' + 1$. If the leader makes a proposal $B'_{k'}$ containing a certificate that does not extend B_k, by the inductive hypothesis, this certificate must be ranked lower than $C_v(B_k)$. Consequently, no honest party in set S will vote for it, and thus a certificate $C_{v'+1}(B'_{k'})$ cannot be formed. Thus, if $B'_{k'}$ was committed and $C_{v'+1}(C_{v'+1}(B'_{k'}))$ is formed it must be the case that $B'_{k'}$ extends B_k. □

Theorem 4.2 (Safety). *Two parties commit the same block B_l for each height l.*

Proof. Suppose for contradiction, two distinct blocks B_l and B'_l are committed at height l. Supposed B_l is committed as a result of B_k being directly committed in view v and B'_l is committed as a result of $B'_{k'}$ being directly committed in view v'. Thus, B_k is or extends B_l and B'_k is or extends $B'_{l'}$. WLOG, suppose $v \leq v'$. If $v = v'$, suppose, $k \leq k'$. By Lemma 4.1, $C_{v'}(B'_{k'})$ must equal or extend B_k. Thus, $B_l = B'_l$. □

Figure 2: The safety proof of HotStuff-2: Optimal Two-Phase Responsive BFT.

to the protocol.

The idea of chaining/pipelining was introduced in HotStuff: BFT Consensus in the Lens of Blockchain to increase the throughput of BFT consensus protocols despite a large number of rounds. Instead of waiting for the view to finish (for the three rounds to finish in our previous example), Chained BFT consensus protocols have each new round start a new view in parallel. We illustrate the optimization in figure 3. This allows each round to not only contribute to their view, but to potentially commit the value of a previous on-going view, reducing the overhead that a large-number-of-rounds protocol can have.

But chained variants of BFT protocols also tend to add additional complexity and are also more susceptible to some liveness attacks (see section 3.6 (Consecutive Bad Leaders And The View

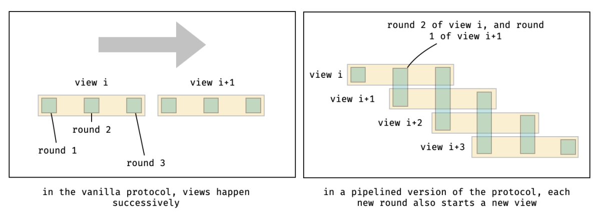


Figure 3: A simplified illustration of the pipelining introduced in HotStuff: BFT Consensus in the Lens of Blockchain.

Synchronization Problem) and section 4.1 ((Distributed) Denial of Service Attacks)).

On the other hand, less fault-tolerant protocols seem to be able to achieve better round complexity. For example, a BFT consensus protocol can achieve agreement in only two rounds if it is willing to use $5f - 1$ participants, tolerating only around a fifth of faulty nodes.

Problem	Timing Model	Resilience	Lower Bound	Upper Bound
BRB	Asynchrony	$n \geq 3f+1$	2 rounds	2 rounds
Psync-BB	Partial Synchrony	$n \geq 5f-1$	2 rounds	**2 rounds**
		$3f+1 \leq n \leq 5f-2$	**3 rounds**	3 rounds [11]
BB	Synchrony	$0 < f < n/3$	2δ	$\mathbf{2\delta}$
		$f = n/3$	$\mathbf{\Delta + \delta}$	$\mathbf{\Delta + \delta}$
		$n/3 < f < n/2$	**sync start $\Delta + \delta$** **unsync start $\Delta + 1.5\delta$**	**sync start $\Delta + \delta$** **unsync start $\Delta + 1.5\delta$**
		$n/2 \leq f < n$	$(\lfloor \frac{n}{n-f} \rfloor - 1)\Delta$	$O(\frac{n}{n-f})\Delta$ [34]

Table 1. Upper and lower bounds for good-case latency of Byzantine fault-tolerant broadcast. **Our new (and nontrivial) results are marked bold.** Under asynchrony, all lower and upper bounds are for BRB. Under partial synchrony, all lower and upper bound are for a new broadcast formulation named partially synchronous Byzantine broadcast we propose (Definition 5). To strengthen all results under synchrony, all lower bounds are for BRB and synchronized start, and all upper bounds are for BB and unsynchronized start, unless otherwise specified.

Figure 4: Table from Good-case Latency of Byzantine Broadcast: A Complete Categorization.

In practice, BFT protocols can get quite complex, and as such trying to reduce the number of rounds or the communication complexity can be a well-worth trade off. As we all know, complex code more easily leads to bugs.

On another note, these thresholds assume a number f of Byzantine actors that is hard to quantify in reality. Who knows really how many participants of a protocol will collude at the same time? Although these numbers seem to often be chosen arbitrarily, we have yet to observe a verifiable "$f+1$ attack" in the wild.

Such $f+1$ attacks in real-world BFT systems happen when too many nodes decide to attack the system together (breaking the safety assumption that only f nodes are malicious). They can be seen as the equivalent to the 51% attacks of the proof of work schemes like Bitcoin. Some systems try to disincentivize such $f+1$ attacks using "economic security". For example, some protocols can "slash" nodes that exhibit faulty behavior, where slashing is implemented by wiping the tokens that a node had locked in order to participate in the consensus protocol (often called "stake" in "proof of stake" protocols).

The question is: how hard is it for the attacking nodes to avoid getting slashed during an $f+1$ attack, and how easy is it to get slashed unintentionally? To our knowledge, only validator mistakes have triggered the slashing of stake so far. As operators of validators are not necessarily aware of how BFT systems work, they can easily violate some rules, for example, by running outdated versions of the node software or by not migrating nodes correctly and not recovering important historical states that would prevent Byzantine behavior. This can make slashing more of a nuisance and a huge monetary risk to users who want to participate in the system.

To our knowledge, most BFT consensus protocols are implemented without slashing[3], which still benefits from an interesting implied economic security slashing mechanism: if users of the system detect an attack they will most likely lose trust in the token, which will, in turn, decrease its price dramatically (this is sometimes referred to as "token toxicity", see The cryptoeconomics of slashing).

3.2 Reconfiguration and Long-Range Attacks

As we will see throughout this paper, the "f faulty nodes" assumption is far from the only assumption that papers make and that implementers have to figure out. In addition, some of the assumptions made by papers are much more problematic as they are implied rather than explicitly stated.

For example, most papers assume that the set of validators is fixed forever. In practice, this is rarely the case. Participants often need to be able to join and leave the committee at some point, rotate their keys, and even revoke their keys in case an accident happens.

[3] notably, Cosmos Zones in the Cosmos ecosystem, as well as Ethereum, have implemented slashing

Thus, protocols are almost always implemented so that they can be "reconfigured", allowing the protocol to periodically agree on a change of validator set. In practice, we have found that reconfiguration protocols are often overlooked if not totally dismissed. Such protocols are important because they can cause safety and liveness issues. For example, DiemBFT's reconfiguration protocol had to enforce empty blocks up to the commitment of the reconfiguration (usually triggered by a transaction) to avoid safety issues (see State Machine Replication in the Libra Blockchain or diem consensus specification)[4].

It is worth mentioning, though, that some papers do attempt to analyze (see Byzantine Generals in the Permissionless Setting) or specify a permissionless setting for BFT consensus protocols (see Algorand: Scaling Byzantine Agreements for Cryptocurrencies and SUI LUTRIS: A Blockchain Combining Broadcast and Consensus).

In addition, validators that leave the validator set (or committee) create a problem: they are still in control of their cryptographic keys. Since cryptographic keys are usually used to authenticate participants in the instantiated protocols this leads to a problem called "**long-range attacks**" (see Winkle: Foiling Long-Range Attacks in Proof-of-Stake Systems) (also called "posterior corruption" and "costless simulation", see Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake) in which old validators who have left the protocol later turn Byzantine (themselves or by selling their cryptographic keys). Thus, a long-range attack allows an attacker who collects enough old keys to fork an older part of the chain and fool a new node that is catching up into thinking that it is being fed the true canonical chain.

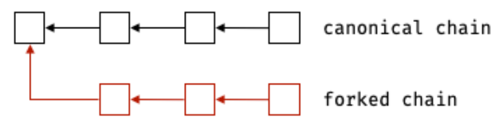


Figure 5: An illustration of a long-range attack where a threshold of faulty nodes manages to collude and rewrite the history of the blockchain.

Long-range attacks are **the elephant in the room** when it comes to BFT elephants. In practice, we can't necessarily assume that validators will always delete their keys when leaving the validator set. A validator is incentivized not to mess with the protocol while they have tokens at stake, not just because of slashing (if implemented), but also because a public fiasco would likely impact the value of their staked assets (they have skin in the game). However, once they leave the protocol, recover their stake, and sell their tokens, they will have zero incentives to play by the book.

Different solutions have been proposed to solve this issue, we discuss them here. Note that solutions that rely on the honesty of nodes (like deleting keys after rotating them) are not discussed here (see section 6.5 (Key Management for Validators)).

The naive solution: Snapshots/Checkpoints. Checkpoints are tuples that uniquely identify a specific point in the history of a blockchain. They usually are implemented as the hash of a block, and come hardcoded into the node's code. As such, they are updated periodically and become a distribution of software

[4] Note that this is related to the pipelined/chained version of HotStuff, which interestingly adds additional challenges when combined with reconfiguration protocols.

issues. As such, most systems rarely rely on them alone and will check other "organic" checkpoints (for example, from blockchain explorers).

A hardware solution: using TEEs. A solution to force validators to only use their keys in legitimate ways was proposed in Securing Proof-of-Stake Blockchain Protocols. The protocol put forward requires validators to use trusted execution environments (TEEs) like Intel SGX and AMD SEV to generate (and thereafter use) their secret keys. To enforce that validators follow that requirement, the TEE must support *remote attestation*, which accompanies any execution output with a signature over the program, inputs, and outputs used (as long as the program was computed within the TEE). This way, validators can register their public keys along with a signature as attestation from the TEE that the private key only exists there, and the rest of the network can verify that. In order to prevent rollback attacks, which an attacker could use to replay queries to the TEE, they rely on replay-protected monotonic counters (RPMCs) that can keep track of the last epoch, view, and round in which a validator has participated, and are protected areas of memory. As the private key only exists in the TEE, the logic in control of it can use the monotonic counters to prevent equivocations and enforce the safety rules of the protocol.

Note that TEEs are not silver bullets. As the signatures they produce come from a hardcoded and hardware-protected key (from Intel or AMD, for example), the trust assumption is reduced to a number of assumptions about hardware security (which hasn't been the greatest in the past[5]) as well as to trust the TEE manufacturer and their supply chain.

A protocol solution: user-based consensus. In Winkle: Foiling Long-Range Attacks in Proof-of-Stake Systems, a protocol is introduced to allow users to individually vote for checkpoints as part of their transactions. Such votes are weighted by users' balances, and a checkpoint is formed when a new epoch (and its validator set) receives a quorum of votes from users. As such, accounts must track their latest signed votes, and the highest checkpoint might lag behind the latest epoch. Nodes still must ensure that the latest checkpoint is not too far from the current epoch, and verify all signatures from all accounts to ensure the validity of the latest checkpoint.

In addition, this system does not seem to be compatible with catchup mechanisms where a node simply trusts the latest state based on a quorum of signatures from the validator set and epoch transitions based on a quorum of signatures from the previous validator set without re-executing and verifying every single transaction. Due to the high throughput of these BFT consensus protocols, nodes often have trouble catching up if they have to re-execute every single transaction, and as such, this catchup mechanism seems to be the most widely deployed one in practice. Although, we note that user-based consensus systems could become practical if augmented with zero-knowledge proofs to allow nodes to avoid having to re-execute everything.

As long-range attacks tend to target new clients that are bootstrapping, or clients that have paused their node for a very long time (which tends not to happen much in practice), organic

[5] See Foreshadow, MDS: Microarchitectural Data Sampling, Plundervolt, and CacheOut: Leaking Data on Intel CPUs via Cache Evictions and SGAxe: How SGX Fails in Practice.

checkpoints (i.e., manually checking that a node's state is the same as other node's after bootstrapping) are often deemed enough in practice.

3.3 Proof of Stake and Weighted BFT Variants

Implementations of BFT consensus protocols often work on top of proof-of-stake systems, that is, systems where participants apply to participate in the consensus by locking (staking) a minimum amount of tokens. This "open participation" is often referred to as a "permissionless" system in the cryptocurrency world, and it usually means that the BFT consensus protocol is augmented with a reconfiguration protocol.

In addition, to handle consensus participants who stake more (which is inevitable), a "weighted" BFT variant is often implemented (which most often counts as a deviation from the paper protocols). In a weighted BFT protocol, the number of votes a participant has is based on their weight, so they can potentially weigh for more than 1 vote.

In this case, an exact quorum of $2f + 1$ votes, for example, might be impossible to achieve as the weights of votes collected in a certificate might sum as either less or more than the threshold. Therefore, code that requires a threshold of exactly $2f + 1$ votes might not work. Interestingly, some BFT protocols specify their cryptographic signature schemes as threshold signature schemes, which amplify these kinds of subtle issues.

This subtlety can lead to liveness bugs in protocols where assumptions are made about quorums that only works when certificates collect an exact number of votes. For example, one implementation we looked at assumed that a quorum certificate of $4f + 1$ votes has a unique "subquorum" of $2f + 1$ votes on the same value. This assumption was violated if the quorum certificate included slightly more than the $4f + 1$ threshold, which led to a liveness issue due to the weighted implementation.

3.4 Fast Processing Against Garbage Transactions

Most of the BFT consensus protocols used in the wild work on a simple chain of blocks, which gets committed chunk by chunk every time a leading block gets committed. That being said, there exist other types of BFT protocols that work on building a large directed acyclic graph (DAG) from different points of view and periodically commit to entire subsets of the DAG at once. For example, Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus, and Bullshark: DAG BFT Protocols Made Practical, which both work in the asynchronous model.

These "narwhal-based" protocols separate the dissemination and ordering of transactions from the agreement/consensus in order to speed up the protocol. The consensus protocol merely has to point to a vertex in the DAG formed in parallel, and peers agree on committing the subDAG under that vertex by doing a deterministic walk.

While this allows the consensus protocol to process a larger amount of transactions, this comes at the cost of potentially redundant transactions. Due to this, it is the responsibility of the application layer to ensure that redundant data does not lead to double spending and is only processed once. In addition, spam attacks could happen where redundant data is sent to multiple validators to fill the DAG in non-optimal ways, leading to commits that commit much less than what they could have. The techniques that such protocols can use to defend against these spam attacks are often vague and understudied. As such, different implementations might use different techniques with more or less efficacy.

3.5 Retrying Messages, A Necessity?

Paper protocols always assume that messages from honest nodes will eventually reach their destination. The time for these messages to reach their destinations is either

- unknown, in the asynchronous model
- or bounded, in the synchronous model
- or eventually the network will get better (for some unknown point in the future) and they will be bounded in the partial synchronous mode

Failing to deliver a message in any of these models could mean that either the sender or receiver is Byzantine, or that there's a (temporary) network connectivity problem. This means that to act as an honest node, one needs to ensure that the messages they've sent are retried and retransmitted until they are acknowledged at the protocol level (as the recipient might have received it but crashed before being able to fully process it). Furthermore, even after recovering from a crash, it might be implied by the protocol that nodes have to understand what messages they need to retry.

The reason for these subtleties in practice is that consensus protocols, as written in whitepapers, often have hidden assumptions about their idealized networking models and the way peers are connected to one another.

3.6 Consecutive Bad Leaders And The View Synchronization Problem

In a consensus protocol, the participant nodes move through a succession of configurations called views (or sometimes rounds). Views are numbered consecutively, and in each view, one replica is seen as the leader. A view is essentially a period during which the leader decides on the next value to agree on and orchestrates the consensus process. The latter is mostly to reduce communication complexity and avoid having everyone broadcast to everyone.

If, during a view, the leader manages to get agreement on a value, it moves to a new view. However, if it fails to help the protocol make progress, the system considers the leader faulty and transitions everyone to a new view as well, but this time with a new leader. This is known as a view change. The goal of the view change mechanism is to ensure that the system can continue to operate correctly even if a leader node is faulty[6].

The leader's role is to ensure that the protocol advances as swiftly as possible during the view they've been elected for. As such, a bad leader can completely stall the protocol and prevent it from making progress, at least for the duration of their view. The view change mechanism is therefore crucial for maintaining the system's liveness and ensuring that it can recover from leader failures or misbehavior, allowing the consensus process to continue smoothly.

Practically, (partially) synchronous protocols im-

[6] where "being too slow" can also count as faulty

plement two liveness mechanisms: leaders are rotated when they fail, and nodes start a timeout when they enter a view. Note that in practice, the rotation of leaders is implemented to automatically happen in every new view (for example, via a round-robin election) as it is good for fairness as well (e.g., to avoid censorship of transactions). If the protocol doesn't advance (from a node's perspective) and this timeout expires, nodes attempt to preemptively enter the next view together. This allows the protocol to move in spite of a bad leader, albeit bounded by a clock. Nodes do this, for example, by sending a warning to the next leader, who can then aggregate them and use a quorum as justification to start a new view.

However, bad leaders can follow one another, and protocols must take that into account (for example, by chaining timeouts and trying leaders from different views). This issue inconveniently faces another issue known as "view synchronization", which must ensure that periods of network instability still end up with enough participants reaching the same view. As we need a quorum of participants in the same view, at the same time, to make progress. This means that timeouts cannot be too quick and that nodes have to wait for each other (at least for the partial-synchronous model).

As such, if timeouts keep on happening, this could either be because of bad leaders (in which case we want to timeout as quickly as we can to "move on") or because of a bad network (in which case we want to increase the timeouts to ensure that people wait for each other within a view).

Successive bad leaders have, non-intuitively, much more impact than f potential successive timeouts. This is because nodes don't necessarily enter a view at the same time, and thus can timeout in staggered ways.

Solutions to this problem are often underspecified and ad-hoc based on non-ideal testing. They generally employ two fixes: nodes will keep on increasing timeouts if views successively timeout (so as to wait for longer and longer periods of time), while timing out preemptively if they see the slightest sign that a single honest node has already timed out in that view. The latter means that at least one honest node had the time to timeout in that view, and can be seen by observing $f + 1$ timeouts.

Note that consecutive Byzantine leaders can have much more impact in chained BFT protocols, as these often require consecutive views to succeed in order to trigger commits. This was discussed, for example, in BeeGees: stayin' alive in chained BFT (Brief Announcement: It's not easy to relax: liveness in chained BFT protocols, BeeGees: stayin' alive in chained BFT).

On the other hand, the techniques discussed previously mostly applied to partially-synchronous consensus protocols and not necessarily to asynchronous ones, which tend to randomly elect their leader in unpredictable ways, and after a set of messages have been collected.

3.7 Leader Election, A Lottery That Should Not Be Gamed

The way the leader gets elected is usually not discussed in papers (or is the subject of entire papers) and, as a consequence, is different in every implementation. Some implementations simply elect leaders in a round-robin fashion, while others attempt to elect the protocol in a more random way, for example, using verifiable random functions (VRFs). Notably, BA★ is known to perform a random election that is not public (and only revealed once a leader publishes their winning ticket).

While round-robin elections can't be gamed easily, they have downsides as they clearly reveal the next sequence of leaders, making targeted DDoS attacks potentially more damaging. Even without an attack, it naturally lets slower nodes have the same probability of being elected as leaders as the more performant ones, which cannot be so desirable either. On the other hand, dynamically choosing a leader can have unintended consequences as it might allow validators to game the system in order to continuously reelect themselves. These kinds of bugs can happen in different ways, as the application layer implemented on top of the consensus often affects the election of the leader. For example, a leader election that uses previous block hashes as randomness to choose the next leader can be gamed by block proposers, or one that orders winners by public keys can be gamed in open systems that let participants register as many new accounts (with arbitrary keys) as they want[7].

A reputation-based leader-election mechanism introduced in DiemBFT v4: State Machine Replication in the Diem Blockchain and then developed in Shoal: Improving DAG-BFT Latency And Robustness and HammerHead: Leader Reputation for Dynamic Scheduling can be used to mitigate the threat of slow leaders. This mechanism involves validators maintaining a record of on-chain scores for each validator and employs a deterministic rule to update the mapping from rounds to leaders based on these scores, prioritizing leaders with higher scores. For validators to agree on the new mapping, they must agree on the scores and the on-chain metrics used to derive these scores. This can be seen as a form of slashing but targeting the liveness of the protocol.

It should be obvious that a leader election that can be gamed can seriously affect not only the liveness of the network, but also the fairness of the network: as a continuously elected leader could decide to censor transactions, or only pick some validators in their quorum messages (potentially affecting the networking score, if there is such a thing implemented, of the ignored validators). Although this is not the case in the Narwhal-base consensus protocols we have talked about for the same reason, consecutive bad leaders were not an issue.

3.8 Byzantine Peers Will Mess With You

Papers often don't specify how peers are supposed to catch up if they're lagging behind. As such, implementations must always ensure that they can receive messages that appear to be "from the future", and that they have a way to retrieve missing information to get to the latest view (and configuration).

There are a number of gotchas in implementing catch-up logic. First, messages from the future are not always legitimate. A node should always expect byzantine behavior from its peers. For example, malicious peers might try to fool you into thinking that you are X views or rounds behind and force you to catch up, only to realize

[7] These attacks are sometimes known as "adaptive key selection attacks" and "randomness-biasing attacks" (or "grinding attack"), see Snow White: Robustly Reconfigurable Consensus and Applications to Provably Secure Proof of Stake.

after quite some time that there was nothing to catch up to.

Simply put, a node should not catch up if there's no reason to catch up. It might be obvious that quorum certificates are a good way to prove to a node that they're lagging behind, but halting whatever a node is doing to go into "catch-up mode" before even verifying the certificate's signatures could be a problem. A certificate is also not necessarily a bulletproof proof (no pun intended) as it might be from a future preceding one or multiple reconfiguration events, in which case the node would not have the latest validator set and their public keys required to verify the certificate itself.

Another example is if you ask a peer for missing information (like a previous committed block), you should make sure that you can handle failure if the peer decides not to respond or respond with bad information. Implementations typically ask other peers, and optionally penalize and/or disconnect from the peer that provided bad information, or was unresponsive.

Finally, "catch-up flows" should try not to disrupt the liveness of the protocol or slow down the protocol in general. There exist trade-offs between delaying catching up and delaying consensus to catch up. For example, If a received message is from the future, but contains an obvious quorum of honest nodes, and the only thing a node has to do is forward it to help drive the protocol to a good outcome, then it might want to forward it first and then enter catch up mode.

4 Beyond Ideation: Navigating the Pitfalls of Instantiation

Now that we've looked at what BFT consensus protocols found in the literature tend to omit (for the misfortune of implementers), let's dive into how they are actually implemented and deployed in practice, and what happens when one takes a paper protocol and instantiates it into a real-world system. We have found that in most implementations we reviewed or worked with, the protocol implemented was actually quite different from the original paper protocols that they originated from. This was due to different reasons.

First, paper protocols are far from being specifications that can be implemented. A lot of aspects are either abstracted, underspecified, or even missing, as we've seen in the previous section. In general, the gap between a paper and a protocol is made smaller through a more detailed specification. Examples of good specifications are RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3, or The Noise Protocol Framework. The goal of these specifications is to be at a level of detail where independent parties could implement the specification in isolation, and their implementations would still be interoperable. Unfortunately, specifications are often less of a priority, and we found that many bugs could have been avoided if a specification had been written. In addition, the writing of a specification often helps simplify the code, its organization, and even the protocol itself.

Second, developers tend to accumulate changes and optimizations, taking paper protocols as unfinished constructions that can be iterated on. This sometimes leads to difficulties in understanding how the implementation works, especially as the paper (or specifications, if they exist) becomes outdated.

4.1 (Distributed) Denial of Service Attacks

Networking issues can lead to catastrophic consequences in Nakamoto consensus-type of protocols because they can fork if enough computing resources exist in the different partitions. For this reason, benign networking issues and more targeted attacks aiming at isolating nodes (called "eclipse attacks") can be very effective both during normal operations and during the bootstrapping of a new node (which we discussed in section 3.2 (Reconfiguration and Long-Range Attacks)).

On the other hand, in partially synchronous and asynchronous BFT consensus protocols, such attacks only tend to lead to liveness issues as the protocols are designed not to fork even in periods of asynchrony (as we discussed above), which implicitly include network partitions. Networking attacks are consequently often studied under "Denial of Service" types of attacks, which we expand on below.

Denial of Service (DoS) attacks are a critical concern in consensus protocols deployed in the real world. These attacks can use bugs in the implementation or inherent protocol flaws. Implementation issues typically involve bugs or vulnerabilities in the code, leading to crashes, loss of connections, or resource exhaustion. On the other hand, protocol issues are fundamental flaws in the algorithm or a misimplementation of a correct algorithm, which can lead to similar liveness issues.

The problem of software bugs is that they are intentionally ignored by paper protocols, which consider them out of scope and implementation detail. Software bugs, at their worst, can allow an attacker to continuously crash the whole network by sending malicious payloads to participants. While this paper discusses a number of such bugs (that can lead to a whole network being attacked), we only discuss how networks can prevent these from going out of control. Although we do mention later in the paper how drastic defense-in-depth solutions could help (see section 6.4 (0days, Faulty Validators and Multiple Implementations)).

Distributed Denial of Service (DDoS) attacks utilize multiple malicious servers (infamously called a botnet) to overwhelm the infrastructure of a node. These attacks are prevalent and pose significant challenges, as in other computer systems. In "An Empirical Study of Consensus Protocols' DoS Resilience" (An Empirical Study of Consensus Protocols' DoS Resilience) different attacks are used to overwhelm validators. Most of the attacks target cryptographic code like signature verification in votes or certificate messages (the latter of which contain many signatures). This is because cryptographic logic tends to be more computationally demanding and can become a target for a botnet to overwhelm validators. In the study, they notice that pipelined protocols are severely affected (as we also mentioned in section 3.6 (Consecutive Bad Leaders And The View Synchronization Problem)), and generally, partial synchronous protocols are somewhat easy to significantly slow down (even without targeting more than f of the participants). They note that, surprisingly, protocols that implement an unpredictable leader do not gain a significant advantage against these attacks (as they can still target the leader once it shows its face). On the other hand, asynchronous and DAG-based BFT consensus protocols like Tusk are much more resistant to these kinds of attacks, as targeting a leader does not stop the protocol from making progress.

Note that these network attacks are often modeled implicitly by asynchronous or partially-synchronous protocols. Asynchronous protocols simply give up on the idea that the network has periods of stability, while partially-synchronous protocols dictate that there can be periods of complete instability as long as they are followed by stable periods of synchrony. These partially-synchronous systems model the time it takes for the transition from unstable to stable as the Global Stabilization Time (GST) but do not explicitly set it to be bounded. However, real-world instantiations do care about minimizing this GST, and as such, DDoS, especially lengthy ones, are undesirable.

There exist several techniques to prevent DDoS attacks, which we will briefly review next.

Protocol defenses. Unpredictable leader elections can help mitigate targeted DDoS attacks as an attacker might not have the time, once they know who the leader is, to perform their attack. There are several methods (Leader Election from Randomness Beacons and Other Strategies) allowing to do that with different levels of safety: Single Secret Leader Election, Ethereum Consensus RANDAO, Algorand's Cryptographic Sortition.

At the same time, those advanced leader randomization mechanisms can be difficult to implement correctly and may require additional cryptographic primitives, which can introduce additional complexity and opportunities for mistakes. In that sense, round-robin leader election is much easier and does not extend the attack surface (and, as said previously, is not gameable). In addition, as "An Empirical Study of Consensus Protocols' DoS Resilience" found out empirically, the efficacy of the unpredictable leader defense seems limited in partially-synchronous consensus protocols, and adopting a DAG-based BFT consensus protocol seems like an interesting approach as it requires disrupting at least $f + 1$ nodes to meaningfully impact the quality of the system (which is the threshold that's taken into account by design in the BFT model).

Protocol setting defenses. BFT protocols are sometimes constructed in a permissioned (also called proof of authority) or semi-permissioned setting, in which validators participating in consensus remain part of the committee for long periods of time. In such systems, IP addresses of the validators don't necessarily have to be made public, and access control is easier to enforce (for example, using a common MAC key shared between the validators).

Topology defenses. Finding a good topology for the network can also help. We discuss this in more detail in section 4.4 (How Network Topology Can Help Validators Defend Themselves).

Software defenses. Many software defenses exist and it is unfeasible to enumerate all of them. One should make sure that the consensus layer (and the application on top) are not doing unnecessary computations, that the protocols used to provide access control are not doing that either, and so on. One could summarize that as reducing the attack surface, reducing the opportunities for slowing down a node and reducing the number of reachable bugs (which is what most of this paper discusses).

Network and infrastructure defenses. Finally, cloud providers are often where nodes are hosted and run, and these usually provide baseline defenses (like IP-based filtering, traffic scrubbing, rate limiting, and web application firewall (WAF) protection) as well as specialized tooling to defend against DDoS attacks (like AWS Shield, Azure DDoS Network Protection, and Google Cloud Armor). However, as these cloud providers are limited in numbers and their features are expensive, this approach can become a source of centralization.

We also generally refer to An empirical study of consensus protocols' DoS resilience (sections 7 and 8) that goes more in-depth on these topics.

4.2 Secure transport: Leader Authentication and Encryption of Sessions

Consensus protocols operate over different types of networking layers that, as usual, provide rich functionalities: node identification and discovery, connection establishment, message broadcast, and node health checking. In theory, consensus protocols just need to be able to have authenticated channels (not necessarily encrypted) and know the public keys of other nodes, which assumes some sort of public-key infrastructure (PKI) designed in the protocol. The most anticipated flaws and vulnerabilities here concern secure handshake protocols or their integration with the consensus PKI model.

Authentication of nodes is often achieved by using signatures, which can be costly to verify (see section 4.1 ((Distributed) Denial of Service Attacks)). Not only that, but processing of other validator messages can also be costly in general. The best practice is to authenticate validators at the peer-to-peer communication level. This is often done by using a secure transport protocol like TLS or the Noise protocol framework. This also has the benefit of encrypting communications (almost for free), which could prevent attacks based on observation of the traffic data.

We have noticed that many protocols chose to use the Noise protocol framework (The Noise Protocol Framework) in place of TLS (mostly due to its simplicity and its numerous security analyses), which can lead to further issues if not used correctly. First, the Noise handshake pattern that is often used is the Noise NK handshake pattern (Noise Explorer), which is subject to two issues: replayability of the first message (facilitating DDoS attacks) and lack of "key confirmation". Lack of key confirmation means that a successful handshake doesn't imply that the handshake was successful for both parties, and this can be addressed by waiting to see at least one message encrypted using the new session key before canceling a previous session. The replayability issue can be addressed by using a counter to each connection attempt (see Noise Layer specification of the Diem blockchain, for example).

4.3 Storage Attacks and Garbage Collection

The short-term/in-memory storage (e.g., some of the messages received from other nodes) and long-term/persistent storage (e.g., safety-critical values like the view number) of a node are often overlooked in papers. In this section we review a few catches that we noticed in real-world implementations.

First, the memory of a node should never grow unbounded. Paper protocols sometimes talk about garbage collection (see Bullshark: DAG BFT Protocols Made Practical) to ensure that the protocol itself doesn't contribute to unbounded growth. However, implementations have many more opportunities to grow their memory to the point of crashing. For example, nodes keep track of many messages and state variables in extendable data structures like arrays or hashmaps in order not to equivocate (i.e., produce conflicting and unsafe protocol values) as well as to prevent other nodes from lying to them (for example, by keeping track of other nodes' last proposals and votes). Ineffective designs and data structures might thus facilitate attacks that aim at slowing down a node by feeding them data that they'll store for too long. Another related implementation detail worth mentioning is that certain languages (like Erlang) do not, by default, randomize the hash function used in their hashmaps. This can, in some scenarios, allow attackers to produce well-crafted messages that collide with other messages stored in the hashmap of a node, effectively introducing inefficiencies in the node's storage and potentially provoking heavy slowdowns.

Second, the storage of a node should never be updated before it knows for sure that this is the right thing to do. For example, it should verify the validity of a peer's message (e.g., its provenance, its signature, its well-formedness, etc.) before updating its storage. In one implementation we looked at, an attacker could spam nodes to increase their memory usage ad infinitum due to the nodes' logic failing to check the validity of messages before updating their storage. This not only gradually slowed down the targeted node in an attack, but it also created an easy way for an attacker to completely halt the network. The attack worked this way: an attacker could produce a malicious payload for a view far in the future and impersonate some validator A using a garbage signature; the invalid message can then be sent to all the other nodes, which will save it in their storage before aborting the processing of the message (due to the invalid signature); but at this point, as they remember seeing a message from validator A in some far away view they would discard any valid message from A up until the future view is reached, effectively banning A from the validator set for an arbitrary period of time. Doing this to enough validators effectively trimmed down the number of active validators under the required threshold for liveness. We show a snippet of the code in listing 1.

Finally, it is important not to prune data when that data is required to be available. For example, in many BFT protocols, signing a proposal means that the node should then be able to serve the data if requested until the data gets committed or garbage collected (if the protocol has garbage collection). If the node loses the data (because of a crash, for example), it might lead to liveness issues as a block gets committed, but everyone has lost access to the block. As such, persisting data and signing/broadcasting the same data often have to be done in that order in order to avoid potential liveness issues. In addition, we have found implementations that wrongly implemented the garbage collection of blocks due mostly to the paper protocol being confusing, specifically the implementation was garbage collecting any block past a certain view whenever a new view was reached, instead of garbage collecting only when a commit happened. The misimplementation could have led to liveness issues in some scenarios for the reason stated previously.

```rust
impl StateMachine {
    pub(crate) fn process_replica_commit(
        &mut self,
        ctx: &ctx::Ctx,
        signed_message: validator::Signed<validator::ReplicaCommit>,
    ) -> Result<(), Error> {
        // TRUNCATED number of checks that will pass...

        // Get current incrementally-constructed QC to work on it
        let commit_qc = self
            .commit_qcs
            .entry(message.view.number)
            .or_default()
            .entry(message.clone())
            .or_insert_with(|| CommitQC::new(message.clone(), self.config.genesis()));

        // If we already have a message from the same validator and for the same view, we discar
        let validator_view = self.validator_views.get(author);
        if validator_view.is_some_and(|view_number| *view_number >= message.view.number) {
            return Err(Error::DuplicateSignature {
                message: commit_qc.message.clone(),
            });
        }
        self.validator_views
            .insert(author.clone(), message.view.number);

        // TRUNCATED...

        // Check the signature on the message.
        signed_message.verify().map_err(Error::InvalidSignature)?;

        // TRUNCATED...
    }
```

Listing 1: Vulnerable snippet of code that verifies the signature over a message received from another validator after storing the message.

4.4 How Network Topology Can Help Validators Defend Themselves

Validators are connected to other validators and need to eventually serve the data they have reached a consensus on to other peers. These other peers, often called clients or users, can't do anything useful if they can't access the data. This sort of represents a departure from the classical BFT literature that mostly focused on distributed databases in private networks, whereas all the more recent research and application has seen its place in the blockchain world.

Of course, users being able to directly query validators represents a liveness risk (as we talk about in section 5.4 (Panicking Or Not?)). To our knowledge, we've observed a few network topologies. They can be categorized in the following ways:

- **No protection**. Clients can directly query validators. This is dangerous for obvious reasons.
- **Infrastructure protection**. A similar topology, except that a validator runs in the cloud and is "shielded" by anti-DDoS services. This is safer, but still subject to DoS attack through protocol and software bugs.
- **Protocol protection**. "Pre-validators" stand in front of a validator's node, terminating connections and either forwarding messages to a validator or serving responses themselves if they can (usually, consensus messages are routed to the validator, everything else can be served as if the pre-validators were replicas). Typically the pre-validators would sit in the same private network as the validator (perhaps using VPC peering) so as not to expose the validator directly to the Internet. These pre-validators can be set up to be much lighter nodes that reuse some of the same validator logic to perform some amount of validation. This approach can limit attacks that target flaws in the software as they would only affect the pre-validators but not validators themselves (and would thus allow validators to continue contributing to the consensus protocol). As these nodes run logic, they are more costly to run. This concept is called "sentry nodes" in Cosmos (Validator Security in the Cosmos Hub) and validator full nodes in Aptos (Node Networks and Synchronization in Aptos)
- **Private network protection**. This is similar to the previous protocol protection, except that validators talk directly to one another via a virtual private network. This is, for example, what Aptos does (Validator Nodes Overview in Aptos). In addition, permissioned networks usually have the luxury of being able to use over-the-shelf VPN solutions like Wireguard.

We recapitulate these different topology configurations in figure 6.

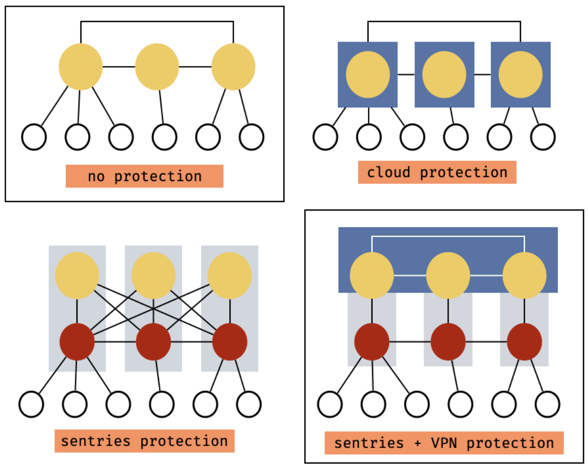


Figure 6: The different network topologies in which BFT validators are deployed.

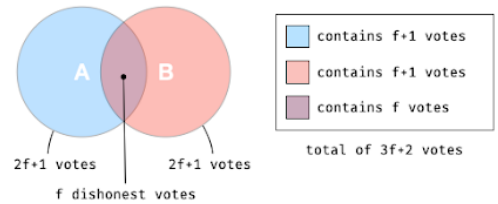


Figure 7: A visualization of an absurd situation where the quorum intersection property is violated, as two conflicting proposals A and B received a quorum of $(2f+1)$ votes

4.5 Imperfect Quorums In Practice

The first instantiation that an implementation of a BFT protocol is faced with, is to figure out the number of participants, which in turn will dictate the number of Byzantine participants that the implementation will be able to defend against.

Let's take the example of $n = 3f + 1$, which is quite common. As said previously, an implementation will typically choose a number of participants first (the n in the equation). But depending on the value n, the equation might not have a solution for f in the natural numbers.

For example, imagine that $n = 100$. Then if $n = 3f + 1$ we have that $f = (100 - 1)/3 = 33$, this works perfectly. But what if $n = 105$? Then $f = (105 - 1)/3 = 34.66\ldots$, what do we do with that? The response is nuanced: one has to make sure that all assumptions used in the safety and liveness proofs are preserved, especially considering how the thresholds are computed in practice.

For example, the **quorum intersection property**, which is used in all BFT consensus protocols, says that two quorums of $2f+1$ votes can't be obtained on two conflicting proposals A and B, which we picture in figure 7.

The property can be proved by showing that the intersections of the honest nodes that voted for both conflicting proposals is not empty (which is absurd as honest nodes are not supposed to vote for two conflicting proposals):

$$\begin{aligned} |A/F| + |B/F| + |F| &= (f+1) + (f+1) + f \\ &= 3f + 2 > n \end{aligned}$$

However, this assertion doesn't necessarily hold anymore when the number of Byzantine participants or the number of participants is obtained by rounding up or down. Keeping our example above, imagine that with 105 participants, we decide to set $f = 34$ faulty nodes. Then, there are two ways to compute the threshold required for a quorum, with one which could potentially lead to a safety violation:

- either we calculate a quorum as $n - f = 105 - 34 = 71$ votes
- Or we calculate it as $2f + 1 = 69$ votes

While $n - f = 2f + 1$ when we have $n = 3f + 1$,

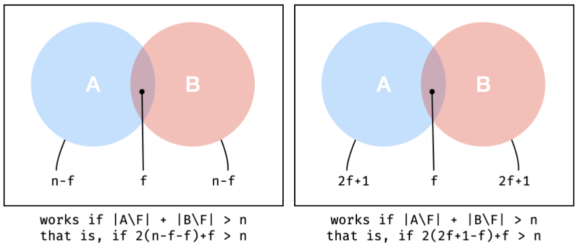


Figure 8: The impact of the threshold computation on the quorum intersection property.

when we don't have that, then we must ensure that the quorum intersection property is true, as pictured in figure 8

In our example, we would have that:

- if we picked a threshold at 71 votes, then $(71-34)\cdot 2+34 = 108 > 105$ and the quorum intersection is still proven correct
- If we picked a threshold at 69 votes, then $(69-34)\cdot 2+34 = 104 < 105$ and the quorum intersection is not valid anymore

In the last case, the careless selection of threshold could lead to safety violations. As such, implementations have to ensure that they compute thresholds in a manner that preserves the correctness of the safety proof.

4.6 Implicit State Transitions Are Hard To Trace

While most cryptographic protocols implemented today are quite linear and sequential in nature, BFT protocols admit many possible state transitions. As such, they are often implemented via an "actor pattern", where a supervisor launches different actors in (potentially green) threads, and where each actor is in charge of processing specific types of messages and events. The supervisor is also in charge of making sure that proper channels of communication are set up between actors. For example, implementations typically have a network actor, a consensus actor, a block syncing actor (for catch-up), etc. The consensus actor itself is often further subdivided into multiple actors, at the very least one for the non-leader part and one for the leader part (that gets busy when the validator enters a view in which they are the leader).

This kind of architecture and design makes state transitions less explicit, and thus it is harder to prevent incorrect state transitions. It would be interesting to see implementations that clearly and exhaustively state all legal state transitions and prevent any state transition from happening if it is not from a whitelist of state transitions.[8]

The complexity of the state transitions of a node impacts, in turn, the analysis of the entire protocol. This makes finding issues in the instantiated protocol's possible state transitions quite difficult as one needs to think about all the different possible states in which the network might find itself. A computational approach to testing different scenarios in which nodes misbehave was attempted in Twins: BFT Systems Made Robust. The research simply iterated through scenarios[9] in which Byzantine nodes were instantiated by running two nodes (unaware of each other) for the same validator identity. This allowed them to easily provoke Byzantine behavior like equivocation (e.g. voting on conflicting proposals) while being able to test the actual up-to-date

[8]This approach can sometimes be seen in more sequential protocols like TLS. For example, Meta's TLS 1.3.

[9]executing 44M scenarios daily, in a fuzzing-like approach (as described in section 6.1 (Static and Dynamic Testing To Find More Bugs)).

node implementations. By testing many different configurations in all kind of network partitions they manage to rapidly find known, as well as new, safety attacks.

Liveness checking of Streamlined Blockchain Consensus extends the Twins framework to detect liveness issues with interesting results. To do that, they extend a node's implementation in order to allow for detection of "hot states" (states that have high chances of representing a liveness issue) as well as live locks (which they call "lassos").

4.7 Detection and Forensic of Byzantine Behavior

When slashing is not implemented, real-world systems often do not do much to detect equivocation and, more generally, Byzantine behavior. The reasoning is perhaps that individual and isolated Byzantine behavior doesn't do much by design, at least up until the tolerated threshold of faulty nodes, at which point it is too late to do anything anyway.

But in practice, there isn't much cost to storing enough information to be able to log Byzantine behavior and investigate why it happened in the first place. It should be obvious that Byzantine behavior in isolation is highly likely to be a legitimate bug that needs fixing rather than an attack. Thus, investigating these could help prevent accidental forks or liveness issues.

Of interest, "BFT Protocol Forensics" (BFT Protocol Forensics) studies the question of "what BFT protocols facilitate the detection of such byzantine behavior?" See also Player-Replaceability and Forensic Support are Two Sides of the Same (Crypto) Coin.

4.8 Instantiating Cryptography: BLS Signatures and Forgery of Certificates

Instantiation of the cryptographic blocks can lead to bugs too. The two main cryptographic primitives that BFT consensus protocols tend to use are signature schemes and (consequently) hash functions. Authenticated encryption can also be used to create secure channels between participants (as discussed in section 4.2 (Secure transport: Leader Authentication and Encryption of Sessions)). In the non-canonical representation section later in this paper (section 5.7 (Non-Canonical Representations)) we discuss how a misuse of hashing led to a bug. In this section, we focus on how we uncovered a misuse of the BLS signature scheme.

The BLS signature scheme is often used in BFT consensus protocols to minimize communication complexity. This is because it supports "signature aggregation," which allows anyone to compress many signatures (often over the same message) to a single signature. This allows validators to save on bandwidth and accelerate the exchange of quorum certificates.

Nonetheless, BLS has its own subtleties and gotchas, it is well-known to be vulnerable to **rogue key attacks** (BLS Multi-Signatures With Public-Key Aggregation) if not used correctly. A rogue key attack allows a malicious actor to forge an aggregated signature that looks like other (victim) signers were involved when they were not.

The first line of defense against such attacks is to ensure that each participant truly knows their keypairs. As discussed in The Power of Proofs-of-Possession: Securing Multiparty Signatures against Rogue-Key Attacks, not any "public key validation" scheme is secure, and either knowledge of the secret key (KOSK) or a Proof of Possession (PoP) scheme (with a separate hash function) must be used. Several of these solutions are standardized in IETF's BLS draft standard (section 3.3).

If the public keys of the validators of a BFT consensus protocol are not vetted in some way that would prevent such rogue key attacks (for example, they rarely are at genesis), it is possible that two malicious validators could collude to forge arbitrary quorum certificates in the consensus protocol. The reason for the collision is that one malicious replica would need to create a malicious keypair that would cancel a quorum of public keys when used to sign. The ill-formed keypair would not allow them to sign on their own, and thus would not be able to participate directly in the consensus protocol without the help of the other malicious replica.

To understand the attack, we briefly recap how BLS works. BLS works by checking that a signature S is equal to $r \cdot x$, where r is an "unknown and hidden value" derived from the message and x is the private key of the signer. The check works because of the combination of:

1. r being unknown and hidden: given a message and a base point P, anybody can compute $Q = r \cdot P$, but not r itself
2. only the signer can produce $(r \cdot x) \cdot P$: they can do so by scaling Q with their secret key

Anyone can then verify the equality without knowing the values themselves by using a pairing over public values (the hidden value $Q = [r]$, the public key $X = [x]$, and the signature S):

$$e([r],[x]) = e(S,[1])$$

An aggregated signature over the same message is simply the addition of multiple signatures $\sum_i S_i$, which can be checked with a similarly aggregated public key $\sum_i X_i = \sum_i [x_i]$. This follows from the generalization of the previous pairing equation:

$$e([r], \sum_i [x_i]) = e(\sum_i S_i, [1])$$

The attack works by creating a keypair $(\tilde{x}, \tilde{X})$ such that its public key cancels a set S of "victim" public keys involved in the aggregated signature:

$$\tilde{X} = (\tilde{x} \cdot P) - (\sum_{i \in S} X_i)$$

and to simply sign as if we were alone:

$$\tilde{S} = \tilde{x} \cdot Q$$

so that the pairing check passes:

$$e([r], \tilde{X} + \sum_{i \in S} X_i) = e(\tilde{S}, [1])$$
$$\Leftrightarrow e([r], [\tilde{x}]) = e([\tilde{x} \cdot r], [1])$$

4.9 Upgradability of the Protocol

Upgrading a protocol is a difficult topic as there's no size-fits-all protocol or pattern that's commonly used. Most protocols do go through upgrades, either to patch vulnerabilities, to improve the protocol, or to change the rules of the system over time. As such, different designs are often observed in the wild.

We have found that most protocols are poorly prepared for flexible protocol upgrades, perhaps due to the fact that, most often, BFT consensus protocols require everyone to update anyway, which allows for clean updates of the protocol, message types, and data structures. Indeed, unlike protocols like TLS that must constantly support older systems that cannot easily update to newer versions, BFT consensus protocols are most often used in scenarios where everyone can agree to update at the same time[10].

Although this is not always the case, the blockchain world has coined the term "soft fork" for an update that will work even if not accepted by everyone. Although, these updates tend to be application-layer updates, which would not work for a BFT consensus protocol that requires as many nodes as possible (and at least a quorum) to be on the same page in order to ensure liveness.

5 General Implementation Concerns

So far, we have only covered protocol-related issues. In this section, we review a number of pure implementation-related issues that we have found to be quite common in BFT implementations, as well as insights that we have learned from working and auditing real-world BFT consensus implementations.

5.1 Graceful Recovery and Persisting Data

BFT protocols often assume that honest nodes will follow specific rules to the letter, which will allow the protocol to remain safe. That assumption is easy to write on paper, but trickier to implement in practice as nodes can, for example, crash at random times.

As such, if a rule says "you shall not vote more than once in the same round" or "you shall not vote for a previous round" or "you shall not propose different blocks during the same round", a node needs to keep track of the rounds it participated in even if it crashes at the wrong time.

In practice, this means two things:

1. That important safety-related data must be persisted, so that if a node crashes, it can recover its latest state and avoid equivocating or doing something not-by-the-book.
2. That this important data must be persisted at the right time, as crashes could happen at any inconvenient place in a node's logic.

The second one is a good source of bugs. For example, a node might produce a proposal, broadcast it, and then persist it. But what would happen if it crashes right after broadcasting it? It would recover from a bad state and equivocate (i.e., broadcast a different proposal).

In addition, a validator should also ensure that they do not end up in a state where the data they have persisted is not enough or inconsistent in order to fully recover from a crash. Inconsistency usually arises when different parts of the system that act concurrently end up persisting conflicting data.

This kind of bug can be found by employing error-injection frameworks during testing[11]. Note also that "graceful recovery" is a field of research that

[10] Notably, due to the nature of these "abrupt updates" that do not need to be backward compatible, the update process of Cosmos was known to be vulnerable to forced malicious update that would result in remote code execution (Not Your Stdout Bug - RCE in Cosmos SDK)

[11] for example, the fail Rust package (crates.io/crates/fail) was used by DiemBFT to randomly inject errors in different parts of the system during testing

led to the creation of Erlang and the "let it crash" philosophy (Making reliable distributed systems in the presence of software errors), but to our knowledge no consensus implementation follows the Erlang philosophy of running different agents that can crash in isolation and easily be restarted by their supervisor.

5.2 The Hidden Cost of Processing Redundant or Unnecessary Data

Serialization (and deserialization) of data structures must be performed for every different type that needs to be exchanged through the wires as part of the consensus protocol. That is, as objects defined in code leave the boundaries of a validator to meet another one, they are encoded in a binary format and then sent over the wire and then decoded or deserialized at arrival by another validator to interpret that as an object in their (potentially different) programming language.

Structures that are being serialized often contain fields that represent cached values that can be derived from the structure itself (for example, the hash of the object). While these fields do not necessarily need to be serialized, as they can be recomputed, they are often found to be serialized anyway. This means that deserialization must be done carefully to ensure that any of these cached values actually contain the correct value.

In general, we have seen these types of bugs leading to unexpected behavior and sometimes devastating issues as it is common to forget to validate these values. In a different context, the developer might think that since these values are redundant or computable from the structure itself, they must be correct.

Some languages like Rust provide macro-based libraries like Serde to generate large amounts of blueprint serialization code, avoiding human errors in manual repetitive code[12]. In addition, this kind of library allows developers to mark fields that should be ignored when deserialization, forcing the correct local recomputation of those. We give an example in listing 2.

```rust
#[derive(serde::Deserialize)]
struct Block {
  author: PublicKey,
  // block_hash will not be deserialized
  #[serde(skip)]
  block_hash: Option<Digest>,
  payload: Payload,
}
```

Listing 2: Deserialization of a Block that ignores the cached computation of the block hash and forces local recomputation.

As a general rule, if some data doesn't need to be sent over the wire, the application should make sure not to send it (and not to deserialize it if it is present). On top of the savings in bandwidth it avoids this class of bugs "by design". Avoiding serialization of redundant fields forces the deserialization to always recompute these fields, which in turn reduces the risk of deserialization bugs where someone forgets to check that these fields contain the correct values.

5.3 Validating Untrusted Inputs

A BFT protocol on paper often dictates the honest way of following the protocol, but not necessarily the exact steps to take to be honest and

[12]This is not an isolated instance of a programming language helping developers to avoid entire classes of bugs by being well designed.

to enforce that others are honest as well. When instantiating a protocol in a real-world implementation, such logic has to be implemented and enforced on both sides of a communication channel and for every consensus message defined in the protocol.

For example, a leader must ensure that they correctly construct their messages according to the rules, but the other validators that receive and parse that message must ensure that it is correctly formed as well (as a malicious leader can do whatever they want). This is true in the reverse scenario as well or between non-leader validators. This duplication of logic can sometimes lead to bugs as implementers can forget to implement both sides. For example, do certificates really contain a quorum of votes? Are messages signed? Are the signers part of the current validator set, and are they only appearing at most once per quorum certificate? Is the author of a proposal the leader of the current round? etc.

This validation needs to happen every time a message is received, but depending on implementations, we have found that they are not always implemented and executed at the same place. This can make it hard to audit such code as we must chase where validation happens. In addition, refactors might remove validation or add important logic that happens prior to the validation (as was discussed in section 4.3 (Storage Attacks and Garbage Collection)).

The type state pattern (The Typestate Pattern in Rust) can help by enforcing validation via the type system, preventing bad code from even compiling. The type state pattern essentially defines objects in different states (validated or not) and gates some operations behind one or the other state. For example, in Rust, this is possible using type parameters, and we illustrate this in listing 3 with signed messages that can be read and processed only after the signature has been validated.

Not every language can support the construction of such abstractions at compile time, but writing runtime abstractions to enforce these same rules can be an interesting pattern to implement, nevertheless.

5.4 Panicking Or Not?

Using panic-like functions inside the core logic of a consensus protocol can halt a validator and cause liveness issues (for example, through targeted DoS attacks, as discussed in section 4.1 ((Distributed) Denial of Service Attacks)). This is particularly critical at entry points where external input controlled by malicious attackers is processed (e.g., public APIs)[13].

As consensus protocols are often implemented as part of larger systems that need to be exposed to users in order to be useful (see section 4.4 (How Network Topology Can Help Validators Defend Themselves)), it is important to remove unnecessary panics that could be reached from the outside and replace them with proper errors being returned to the users (and potentially log them) instead. Panics should be reserved to important assertions in places where a violation could lead to undefined behavior (i.e., better crash than not understand what would happen if we let the logic continue).

Using fuzzing techniques (which we discuss in section 6.1 (Static and Dynamic Testing To Find More Bugs)) on the Libra/Diem codebase, we

[13] See ABCI methods panic.

```rust
pub struct Signed<V: Variant<Msg>> {
    /// The message that was signed.
    msg: V,
    /// The public key of the signer.
    pub key: validator::PublicKey,
    /// The signature.
    pub sig: validator::Signature,
}

impl<V: Variant<Msg> + Clone> Signed<V> {
    /// Verify the signature on the message.
    pub fn verify(self) -> anyhow::Result<V> {
        self.sig.verify_msg(&self.msg.clone().insert(), &self.key)?;
        Ok(self.msg)
    }
}
```

Listing 3: Using a typestate pattern in Rust to prevent code handling unsigned messages.

found many issues of the `unreachable!()` macro in Rust, a seemingly benign function meant to indicate parts of the code that cannot be reached and panics to indicate that the assumptions were false. For every actually-reachable instance we found, the panic was unnecessary and could have simply been replaced with an error to the user or a log to indicate weird node behavior needing investigation. In addition, we used linters to detect code that could panic when it didn't need to panic (for example, `TryFrom` implementations in Rust are already expected to potentially return errors, and thus should not contain panics).

Note that it is not always straightforward to understand what code can panic. For example, functions of the standard library (or other dependencies) might not properly document if they can panic. (For example, we found panics with usages of the Rust `copy_from_slice()` function, which at a glance could seem infallible.)

5.5 Integer Overflows

Integer overflows are another potential opportunity for errors when implementing consensus protocols, as paper protocols never talk about the serialization and encoding of their messages and types.

One common notable issue is to choose data types that are too small for a long-term run of the protocol. Depending on the speed of the network, a 16-bit data type to count the number of blocks, views, etc., might not be enough, while a 32-bit data type could be enough. (Imagine that a BFT consensus protocol could achieve one view per second, then it would require 136 years to overflow a 32-bit data type.) That being said, we usually recommend using 64-bit data types just to be sure.

Another issue is to gracefully handle underflows, especially when the protocol starts (or restarts). For example, reaching out for previous rounds when the round number is 0 will not work. In these edge-cases, it is important to not produce negative values or underflow[14].

One might wonder if crashing on under/overflow is a good idea. This can be a bit too extreme as operations overflowing can happen in all sorts of benign places (including logs), which would transform a number of non-issues into liveness issues. In addition, this adds quite an overhead to all operations that happen in a node implementation. This is most likely why most popular languages do not panic on overflows[15].

Note that in our experience, most integer overflows found in consensus protocol implementations were non-impactful.

5.6 Non-Determinism and Concurrency Issues

Non-determinism in validator's logic can lead to validators failing to come to a consensus due to failing to take the same actions or failing to come up with the same values (for example, a value to agree on).

One way non-determinism is exhibited by consensus protocols is in the implementation of a node which often involves multiple threads and concurrent tasks (as mentioned in section 4.6 (Implicit State Transitions Are Hard To Trace)). As concurrent code provides no guarantee on the ordering of its concurrent tasks (which is often affected by the network and protocol timeouts), concurrent code can lead to concurrency issues in which nodes exhibit unsafe behavior (e.g., equivocation) or dangerous liveness behavior (e.g., deadlocks).

The paper "Concurrency Testing of the HotStuff Distributed Consensus Algorithm" investigated using different techniques to produce different program traces, exercising different ordering of concurrent tasks to potentially find such concurrency bugs. They obtained good results, but as far as we're aware, this requires specialized tools (e.g., Concurrency Unit Testing with Coyote) that might not work with every stack.

In other situations, the implementation itself directly makes use of non-deterministic **data structures**. For example, hash tables (also called hash maps, dictionaries, or associated arrays) are often implemented in programming languages in such a way that iterating over them returns their entries in a different order. This is usually because of the randomization of their internal hash function (for security purposes). Thus, if a consensus protocol requires validators to all pick the same entry out of several entries of a hash table, they might fail to agree on the one that they picked.

Another bug we found was a bug in a sidechain that used Tendermint Core in a service-based model instead of the traditional ABCI-based model for ordering input transactions. In this design, each sidechain node interacts with a trusted Tendermint node using its RPC. A sidechain node sends input messages to the corresponding Tendermint node. Tendermint validates these messages and ensures they are recorded on all Tendermint nodes in the same order within blocks. The sidechain node then retrieves Tendermint blocks via Tendermint RPC, performs static

[14] some languages support augmented arithmetic functions like `saturating_sub()` in Rust, which will return 0 instead of negative or large values when an underflow occurs.

[15] Although some languages like Solidity have decided to do so due to the number of devastating bugs that were happening due to overflows.

and semantic validation, removes corrupted messages, generates a block with the verified messages, and applies them directly. Since Tendermint nodes did not perform semantic message validation within ABCI, Tendermint blocks may contain corrupted (malicious or invalid) messages. Meanwhile, the sidechain mempool performed all necessary input validation checks. To address this, message filtering for all Tendermint blocks was implemented: if a Tendermint block contained corrupted messages, they were filtered out before the block was sent to the sidechain node for execution. The bug occurred because of filtering, not all valid messages were included in the corresponding sidechain block due to Golang's range iteration. As a result, this led to a safety violation.

5.7 Non-Canonical Representations

A lack of canonical representation for a protocol object can also lead to non-determinism or confusion bugs where a value is interpreted as a different one by different nodes. Such a flaw could lead different validators to interpret the same object in different ways or different objects in the same way.

A good example of that is in **malleable DAG representations bugs**. A DAG can be seen in two ways:

- Either, with vertices made out of blocks, and edges implying the existence of "quorum certificates", allowing them to extend other blocks
- Or, with vertices made out of certified blocks, explicitly encoding the quorum certificates in the vertices themselves (or in the graph)

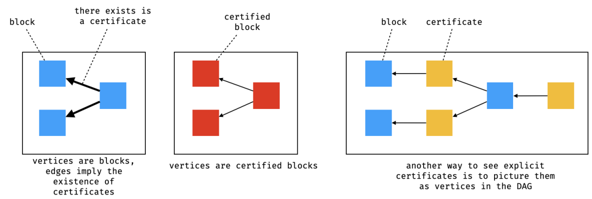


Figure 9: The different ways of encoding or interpreting a DAG, visually.

The problem with one of these representations is that **certificates are malleable**: any combination of signatures works as long as they reach a quorum. As such, different validators trying to come to a consensus on a DAG might not agree on the DAG if they look at the second way of representing the DAG (see figure 9).

Due to this, some implementations can "lock" themselves in an irreparable state because they can't retrieve an updated DAG from another peer that is compatible with their local DAG.

But at worst, non-canonical representations can lead to safety issues as well. For example, if a hash function does not uniquely identify a block, due to a **non-injective encoding**, it might lead to the logic being applied to incorrect or malicious data. We found such a bug in a cryptocurrency application in which the implementation did not encode blocks in a bijective way. This led to validators being able to lie to others about the content of a block that they were agreeing on. To understand how this can happen, imagine the following example that uses a naive encoding to hash a transaction of 100 tokens from Alice to Bob, with 15 tokens of fee.

As this transaction is non-injective, one can easily shift values around to produce a different transaction (here with a transfer value of 1001 tokens and 5 tokens of fee) that hashes to the same digest. We illustrate this in listing 4.

Another example of a non-canonical representation bug is due to malleable signatures. Most signature schemes are malleable (On the Malleability of ECDSA Signatures , Taming the many EdDSAs) in the sense that either the signer can produce a different valid signature for the same message, or an observer without control of the signing key can alter the signature so that it is still valid for the same message. Any protocol relying on the uniqueness of a signature would thus have issues. While, in general, it is bad practice to rely on a signature being unique, protocols that insist on relying on the uniqueness of signatures still have the option to use signature schemes that offer this property! The BLS signature scheme is a good example of such a scheme.

6 Defense in Depth

Nobody writes bug-free code. If someone were to do a study on what causes the most code per line of code, it would probably be a mix of what language is being used (some languages like C are much harder to write correct code with), how complex the logic being implemented is, and how self-contained and simple the code is. BFT consensus protocols tend not to fare well for the last two metrics and can, therefore, hide a lot of bugs. On top of that, implementations are often not just code written by core developers, but an entire tree of dependencies (e.g., libraries, frameworks, etc.) used by the project. The question then becomes, how should one find the bugs? And what can one do to limit the impact of such bugs when they inevitably happen?

6.1 Static and Dynamic Testing To Find More Bugs

We've talked about the benefits of writing specifications and of manually reviewing code, but a lot of automated tools exist to find different classes of bugs.

Static analysis, which comprises tools that analyze code semantics without executing it (sometimes having access to the compiled AST), is often easily accessible to developers through the form of code linters (depending on the programming language). In our experience, these tools are rarely helpful to find meaningful bugs.

Dynamic analysis, on the other hand, includes tools that execute code in order to find bugs. One of the most notable dynamic analysis tools is fuzzing, which has shown to be great at finding tons of memory corruption bugs in C-like programming languages. Unfortunately, most BFT implementations are written using modern languages (like Rust or Golang) that do not offer great opportunities for memory corruption bugs. This does not mean that fuzzing is completely useless, and in our experience, it was useful in finding logic that led to crashes (as we mentioned in section 5.4 (Panicking Or Not?)).

Fuzzing works by instrumenting the program binary and detecting when new inputs lead to taking new paths in the program execution. A fuzzing test usually targets a single function that expects a byte string as input and then uses different heuristics to generate an infinite number of random-looking byte strings to feed to the program. When an input leads to a new path, it then attempts to create more similar-looking inputs to fuzz that path further.

As noted, fuzzing has a pretty simple interface

```
$ echo -n "Alice""Bob""100""15" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
$ echo -n "Alice""Bob""1001""5" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

Listing 4: An example of hash collisions due to ambiguous hashing of the underlying data.

at its core that targets functions that take a single-byte string as input. This makes it a bit of a challenge to understand how to fuzz a BFT consensus protocol, as these protocols offer many different state transitions and entry points that can be fuzzed. Hanno Böck is well-known for having demonstrated that complex cryptographic protocols are still great candidates for fuzzing, showing that the infamous heartbleed bug could have been found on OpenSSL using fuzzing[16]. Since then, we have also used this strategy to fuzz other TLS implementations[17].

We have replicated the same strategy to fuzz consensus protocols by creating different scenarios, in which we set up a validator to receive a specific message and fuzz that message. For example, fuzz a proposal and send it to a validator who expects a proposal[18].

There are a number of things to consider when fuzzing. All determinism has to be removed, as it can trip fuzzers into thinking that they found new inputs leading to new paths, when they have not. This means that async code and threaded code should be avoided, random numbers should be made deterministic, and so on. (Even the fact alone of initializing some state can trip fuzzers, as they think that the next run after the initialization run just found a new path.) Cryptographic checks often can't be faked, and so signature checks have to be removed as well. Fuzzers also need to be as fast as they can be in order to find bugs (as it can sometimes take months to find one bug), which means, for example, that any kind of timer should be removed as they will slow down fuzzers.

Overall, the problem with fuzzing is that setting up and implementing these scenarios is costly, optimizing a fuzzing test takes time, and running them continuously requires infrastructure resources. The scenarios to fuzz BFT consensus protocols are also limited, as a lot of the state transitions are implicit and numerous, and it is difficult to cover all of them. The amount of setup necessary for a node to be fuzzed is also sometimes a blocker (modules need to get mocked, threads need to be avoided, etc.), which can force us to limit ourselves at fuzzing internal functions instead of more general entry points, which potentially misses on covering important logic.

In addition, fuzzers are quite dumb in nature, and their heuristics often fail to efficiently find new interesting inputs. Note that there exists a promising new branch of fuzzing, whitebox fuzzing that attempts to be more clever at finding new paths by using SMT solvers (Billions and Billions of Constraints: Whitebox Fuzz Testing in Production), and even LLMs (White-box Compiler Fuzzing Empowered by Large Language Models).

[16]See How Heartbleed could've been found (2015).
[17]See Fuzzing picotls.
[18]See how fuzzing consensus messages for Libra was implemented

Another option is to use property tests, which were introduced and popularized with the QuickCheck library in Haskell. Property testing libraries share a lot of the same logic with fuzzing libraries, except that they are usually much better integrated and can more easily create random-looking custom structures (and not just byte strings). In addition, property testing doesn't require a (sometimes costly) infrastructure to run, it just runs when tests are run, and any bad input found is saved to run as a regression test in the future. Property tests are often much dumber than fuzzing, though, as they just generate random objects without real strategy. But as we said earlier, fuzzers are still dumb (at least without the whitebox fuzzing approach), and as such the question is: is dumb better than dumber? And do we find most fuzzing bugs after a long continuous fuzzing period? In our experience, it is not clear. As such, we recommend using property testing and augmenting it with fuzzing if one can. And if one really can, then, of course, consider looking into white fuzzing.

6.2 How Formal Methods Help, And How They Don't

Formal verification tools allow a developer to write a formal specification (in some formal specification language, like Dafny, TLA+, Quint) and use a model checker to produce absolute proofs of statements or proofs of why a statement is false. More evolved formal verification tools can also verify the correctness of an implementation (compared to its formal specification) and even generate formally verified code in some cases.

Several efforts on formalization have been recorded using different formal verification tools. DiemBFT had Coq and Boogie formalization, HotStuff with Ivy and TLA (see Formal Verification of HotStuf), ZooKeeper used TLA+, Cosmos/Tendermint used TLA+ and Quint, QBFT used dafny/Boogie, Stellar used Isabelle/HOL (see On the Formal Verification of the Stellar Consensus Protocol), Matter Labs used Quint for their consensus protocol (see Specification and model checking of BFT consensus by Matter Labs).

That being said, there are several problems with using formal methods on BFT protocols.

Safety only. One problem is that formal verification efforts usually tend to target proving the safety of the protocols since it is easier for verification tools[19]. This is because the liveness property is much harder to define, model, and verify (especially for state-of-the-art consensus protocols). As the safety proofs are often easy to prove and verify on paper, this limits the impact of these formal tools.

Divergence from the implementation. Another problem is that a BFT consensus protocol often gets modified quite heavily as it gets implemented. Due to that, the formal specifications tend to get outdated quite rapidly. Note that this is also a problem with non-formal specifications. One partial solution to this issue is to perform conformance checking, by taking traces of the actual protocol implementation and checking that they are legitimate according to the latest formal specification. This approach has also led to finding bugs in real-world systems; see Validating Traces of Distributed Programs

[19] Although there is some line of work in trying to formally prove liveness in some scenarios, see A short counterexample property for safety and liveness verification of fault-tolerant distributed algorithms and On the Formal Verification of the Stellar Consensus Protocol

Against TLA+ Specifications.

Formal specifications don't match reality. Formal specifications, due to their high-level nature, must forgo and ignore a number of implementation details. For example, cryptographic primitives must be modeled as black boxes, network quality can be slightly altered using tricks, state transitions have to be modeled in specific ways, etc. At the end of the day, a formal specification often targets a protocol that looks very different from the actual implementation being analyzed. In addition, as formal specification languages can be quite strict, formal specifications of BFT protocols often accumulate a pile of hacks to make things work. So much so that human mistake can lead to unrelated statements being proven,[20] and the formally verified protocols are often too different from their real-world counterparts to truly provide value, we note that this is not necessarily true if lower-level formal tools are used that are closely tied to the code running in production. For example, this is what AWS has been doing to continuously prove that their TLS protocol is correct in Continuous Formal Verification of Amazon s2n.

That being said, formalization is still an interesting tool to reach for in order to increase guarantees in the analyzed protocols, and formal verification has still shown to be of use to help developers get a better sense of the protocol they're implementing, sometimes allowing them to find bugs without having to run a model checker. This exercise often has the additional side effect of heavily simplifying the protocol (see Applying Formal Verification to Microkernel IPC at Meta, for example), as a developer is forced to find simpler ways to model their protocol in order to write them in the strict specification languages. In that sense, the exercise is sometimes more fruitful than the actual result. In addition, in larger teams, it allows non-developers to get a chance to analyze the protocol at a higher and more rigorous level, without having to spend time in the codebase.

[20] In 2017 the KRACK attack on WPA2 took advantage of a wrong assumption made by the formal proof.

6.3 A Trusted Computing Base To Minimize Compromises

The idea of the Trusted Computing Base (TCB) is to draw the boundaries of the critical components of a system and to "air gap" that part of the logic and infrastructure as much as possible from the rest of the system. This conceptual approach can allow developers to take a more "intense" security approach with the TCB, while allowing the rest of the system to be less strict with its development. The deployment of the project can also physically separate the TCB from the rest of the system, for example, by using a different server or a trusted execution environment (TEE) to run the TCB logic, or by writing the TCB logic in a formally verifiable language.

For a BFT consensus protocol, a natural way to implement a TCB is to take the state and the logic that preserves the safety of the protocol (from a node's point of view) and segregate it from the rest of the node's logic. As we previously mentioned, the rules that a node has to follow and those involved in the safety proof are often limited, which fits nicely with the limited environments offered by TEEs.

This TCB concept can help limit the impact of an exploitable bug in the rest of the codebase (which should have more opportunities for such bugs as it should be much larger than the TCB).

Indeed, an exploit in the rest of the codebase would not allow the node to behave in a way that would facilitate an attack on the system's safety (for example, by equivocating).

An example of this approach is how Diem proposed different designs for a TCB that would minimize trust on other non-TCB components (see A Minimal Trusted Computing Base (TCB) for Diem).

6.4 0days, Faulty Validators and Multiple Implementations

While we know that all BFT consensus protocols can tolerate up to f faulty validators, who these f faulty validators are might dynamically change over time depending on the adversary model used to design a specific BFT protocol. Some models might be secure only when the f faulty validators are fixed in advance, while others might allow for an attacker to decide and update who these f faulty nodes are over time (as long as there are never more than f faulty nodes).

But how do these models really reflect reality? Or even, how easy is it to break the f faulty node assumption? Paper protocols completely ignore the high potential of software bugs, and for a good cause: it would not be a very interesting discussion to theorize a model in which all nodes can be made faulty at the same time. But in the real world, so-called 0days happen every week. 0days (or zero-days) represent the concept of an exploitable bug that has no patch available, making a program vulnerable for some period of time (up until the bug is fixed and the fix is deployed on enough nodes).

If all validators run the same implementation of the consensus protocol, then such a 0day can completely destroy all BFT assumptions we have had so far. For example, the 0day could be a remote code execution attack allowing an adversary to take control of any nodes they want.

The most effective preventive technique to protect against this scenario is to have a diversity of validator implementations. For example, Ethereum encourages developers to implement new implementations of the Ethereum node and encourages users to use different implementations to improve the resilience of the network. Today, clientdiversity.org lists six different Ethereum consensus client implementations, written in different programming languages (Rust, Golang, Java, Typescript, Nim) with different security guarantees, as we show in figure 10.

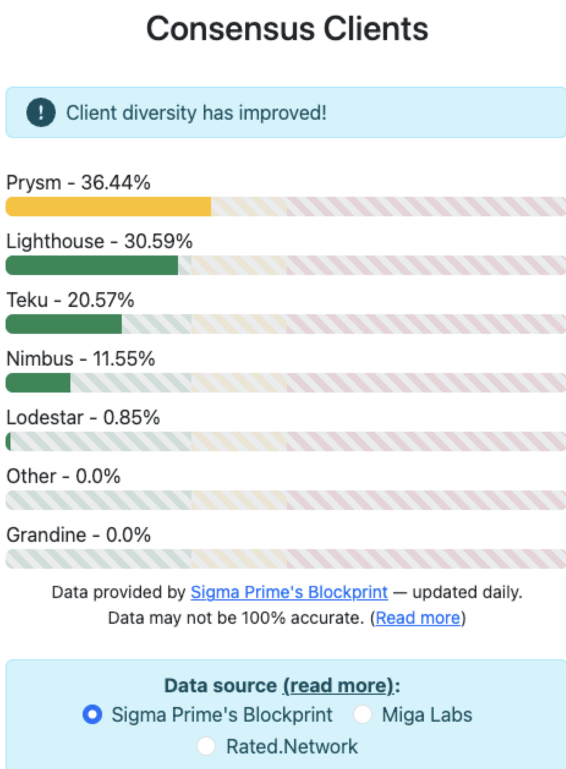


Figure 10: Diversity of Ethereum consensus clients and their current usage, taken from clientdiversity.org on July, 2024.

As such, a 0day found in one of the implementations would only allow an adversary to control a threshold of nodes equal to the number of nodes that were using the vulnerable implementation only. It is thus important that no implementation is used by more than a third of the validators.

In addition, many validators are run in cloud environments (like AWS, GCP, and Azure). It is important that the compromise of a Cloud does not affect the tolerated threshold of Byzantine validators, this is why validators should make sure to not all use the same Cloud provider. More generally, 0days are not just bothersome when found in the node implementation, but really anywhere in the environment running the node (e.g. the Operating System, the file system, etc.). Lazarus: Automatic Management of Diversity in BFT Systems proposes a way to manage the deployment and environment of the different validators in order to maximize the diversity of environments and decrease the chance of a 0day impacting too many validators.

6.5 Key Management for Validators

Different blockchain implementations and different validator operators address the question of where to store keys differently. Naive implementations tend to store private keys in plaintext files, which easily turns a compromise of the host running the node into a leak of the private keys, but can also lead to accidentally pushing the key to a remote GitHub repository (for example). More advanced methods include using specialized hardware to store the key, such as hardware security modules (HSM), so that keys cannot be extracted following a compromise of the node. In general, many ways exist to increase the security of a node's keys, and key management is generally out of the scope of this paper.

More in-scope are software solutions integrated into the protocol that attempts to improve the security of nodes. In general, the protocol solution to decrease the impact of key compromises is to enforce key rotation of the participants often. Key rotation adds a property called forward secrecy (or forward security), which can be found in other cryptosystems like random-number generators (see SoK: Security Models for Pseudo-Random Number Generators) and secure sessions (see Formal Analysis of Session-

Handling in Secure Messaging: Lifting Security from Sessions to Conversations). Forward secrecy means that a compromise at some point in time only compromises the security of the protocol at that point, but not in the past, which is useful against long-range attacks, as we explained in section 3.2 (Reconfiguration and Long-Range Attacks).

In Pixel: Multi-signatures for Consensus, an aggregatable (like BLS signatures) and forward-secure signature scheme is introduced to allow for more fine-grained forward security: key rotation can be enforced at each new message signed. To integrate this efficiently in the protocol, the scheme uses hierarchical ID-Based Cryptography (HIBE) (see, for example, Hierarchical Identity Based Encryption with Constant Size Ciphertext) which relies on pairings (like BLS signatures) to allow identifying validators via a single main public key, from which specific per-message public keys can be publicly derived. Validators then generate all of the associated private keys they will need and can delete each of them after use. A compromise would thus only affect the section of keys that haven't been used yet. Note that the scheme still relies on the honesty of the nodes running it, as they still need to delete keys after use.

7 Conclusion

We surveyed the different challenges developers encounter in the real-world instantiation and deployment of BFT consensus protocols. Many of these challenges were due to paper protocols often missing important key protocols to make them practical (e.g., safe automation of validator set updates), a lack of detail needed to correctly implement these protocols (as opposed to detailed specification and Internet standards available for other types of cryptographic algorithms), or the pragmatic, real-world challenges that one must face when productionizing, distributing, and deploying high-risk software.

While papers often consider the macro, answering questions like "How many Byzantine nodes can we tolerate if they decide to collude?" Real-world protocols must consider the micro: how to avoid software, infrastructure, and protocol vulnerabilities that could let an attacker compromise all or any of the nodes instantly.

As we showed, there is no single silver bullet. Developers must be careful when deploying BFT consensus protocols in production, and remain aware of past bugs and pitfalls in the different BFT consensus protocols and implementations. Detailed specifications should always be written, code should be thoroughly tested using different techniques, and many eyes should review the code.

BFT consensus protocols have shown to be great ways to securely distribute databases in the face of malicious nodes, but more research and development work is needed in order for more users to benefit from secure BFT consensus protocols and their implementations.

Acknowledgements

We want to thank Alberto Sonnino, Igor Konnov, Ethan Buchman, Eleftherios Kokoris Kogias, and Zekun Li for their feedback on an earlier draft of this paper.