Testing Byzantine Fault Tolerant Algorithms

Evaluating the correctness of Tendermint protocol using ByzzFuzz

Antoni Nowakowski[1]

Supervisor(s): Dr. Burcu Kulahcioglu Özkan[1]**, João Miguel Louro Neto**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 26, 2025

Name of the student: Antoni Nowakowski
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Kulahcioglu Ozkan, João Miguel Louro Neto, Dr. Jérémie Decouchant

Abstract

The reliability of Byzantine Fault Tolerant (BFT) consensus protocols is critical for the robustness of modern distributed systems, i.e., in blockchain technologies. Testing of BFT protocols is crucial, as consequences of faults in their implementation can lead to malicious users exploiting vulnerabilities, resulting in financial losses, data corruption, or system unavailability. Such incidents, as seen in real-world attacks on blockchain systems, underscore the need for rigorous testing methodologies to ensure protocol correctness and resilience under adverse conditions.

This paper evaluates the implementation of the Tendermint protocol in the ByzzBench framework using ByzzFuzz, a testing approach for BFT consensus protocols. ByzzFuzz introduces structured mutations to simulate real-world fault scenarios, enabling the identification of incorrect behavior. The main question addressed in this study is: Can ByzzFuzz detect subtle protocol faults more effectively than baseline testing methods, and how do mutation strategies influence fault detection performance?

Through extensive testing, ByzzFuzz successfully uncovered violations in the Tendermint implementation, demonstrating its capability to detect subtle protocol faults. A comparative analysis with baseline testing methods revealed that ByzzFuzz provides greater fault coverage, identifying nuanced issues that the baseline approach missed. Furthermore, the study evaluated the effectiveness of small-scope and any-scope message mutations, where they change a value incrementally and arbitrarily respectively. This study found that small-scope mutations perform better in finding faults.

1 Introduction

The increasing reliance on applications such as online banking, e-commerce, and blockchain-based cryptocurrencies necessitates highly available systems capable of maintaining uninterrupted, accurate service. One solution is the use of replicated systems that rely on consensus protocols to enable agreement on a shared state, even in the presence of faults or malicious actors. Over the years, numerous Byzantine Fault Tolerant (BFT) protocols have been developed to address these challenges, ensuring continued functionality despite a fraction of faulty or malicious nodes.

Blockchain technology exemplifies the critical role of BFT protocols, with consensus mechanisms like Proof-of-Work (PoW) and Proof-of-Stake (PoS) underpinning their operation. While PoW has been a cornerstone for early blockchain systems [1], its environmental impact has driven the adoption of PoS, which offers a more sustainable alternative [2]. Among PoS-based protocols, Tendermint has emerged as a prominent solution, powering systems such as Cosmos Hub. Tendermint, however, like any other consensus protocol is not immune to vulnerabilities stemming from implementation issues. For example, a Binance cross-chain hack leveraging Tendermint-powered systems resulted in the loss of 100 million USD [3], illustrating the risks of protocol implementation flaws.

Despite decades of development, consensus protocols implementations remain susceptible to violations. Issues such as agreement failures and liveness violations continue to surface, even in mature implementations. For instance, termination violations in the Ripple XRP Ledger Consensus Protocol were discovered years after its deployment [4]. These examples demonstrate the need for rigorous testing to uncover hidden vulnerabilities, as undetected flaws can lead to significant security and financial consequences [5].

This research aids in addressing the gap in reliable automatic testing frameworks, by evaluating the effectiveness of ByzzFuzz [4], a fuzzing-based testing approach, in uncovering violations in BFT protocols. ByzzFuzz introduces structured mutations to simulate diverse fault scenarios, enabling the identification of liveness and agreement violations. This study also benchmarks ByzzFuzz against a baseline randomized testing method to assess its relative performance in detecting faults under controlled conditions and compares the performance of different mutation types in detecting violations.

The remainder of this paper is organized as follows: Section 2 provides the necessary background on consensus protocols, the main research questions, and the methodology. Section 3 details the contributions we have made for this paper, including the literature review as well as the description of necessary changes to implement Tendermint in ByzzBench. Section 4 presents the experimental setup and results, followed by a discussion of their implications. Section 5 discusses the ethical aspects of the research conducted in this paper. Finally, Section 7 concludes the paper and Section 8 outlines directions for future research.

2 Problem Statement

This section presents background information to understand the problem, methodology, and terminology relevant to this research. The first subsection provides an overview of consensus protocols and their significance. This is followed by the research questions and the methodology.

2.1 Consensus Protocols

Consensus protocols have been instrumental in shaping modern distributed systems by solving a critical challenge: enabling machines in a network to agree on some data. This ensures synchronization across multiple nodes. For example, consider a banking application where millions of transactions are processed daily. Servers distributed across different parts of the world must provide reliable service. Without consensus protocols, ensuring that all servers agree on the outcome of transactions would be nearly impossible.

Two key properties underpin consensus protocols:

- **Agreement**: "No two nodes in the system can decide differently" [6]. If violated, issues like double spending of a customer can occur, where one node records

a transaction as complete while another erroneously reflects that the funds are still available.

- **Liveness**: It is defined as "something good must eventually happen" [6]. In the context of this research, it can be characterized as ensuring that all the nodes can still execute an action. Considering the banking app example mentioned earlier, this would mean that all valid transactions eventually get processed and recorded across all servers. Even if some servers experience temporary delays or network issues, the protocol ensures that the transaction is eventually confirmed, allowing the customer and merchant to proceed with confidence.

Consensus protocols prevent such violations by ensuring all nodes agree on the validity of transactions, even in the presence of failures or conflicting data. While this example focuses on financial systems, consensus protocols underpin various applications, including e-commerce platforms and social networks.

2.2 Research Questions

In order to determine the performance of ByzzFuzz in testing BFT protocols, the following questions had been formulated.

RQ1. Can ByzzFuzz find any bugs in the implementation of Tendermint?

RQ2. How does the bug detection performance of ByzzFuzz compare to a baseline testing method that arbitrarily injects network and process faults?

RQ3. How do small-scope and any-scope message mutations of ByzzFuzz compare in their performance of bug detection for Tendermint?

By answering these questions, this study provides insights into the performance of ByzzFuzz in testing BFT protocols, with a focus on Tendermint. The findings additionally contribute to the broader goal of creating a platform which allows for different testing approaches to be compared and for benchmarking the performance of various implementations of protocols.

2.3 Methodology

Process

To address the research questions, the Tendermint protocol from Buchman et al. [7] was implemented in the ByzzBench framework, which is a tool which facilitates the testing and allows for benchmarking of BFT protocol implementations with different testing approaches. This implementation was then subjected to various configurations to test its behaviour and identify potential violations. Tests were conducted using multiple approaches, including baseline and ByzzFuzz, to simulate different fault scenarios. The results from these tests were collected and analysed to evaluate the protocol's performance and to derive insights that answer the research questions.

Tendermint

Tendermint is a BFT consensus protocol designed to address the high energy costs of blockchain security mechanisms like those in Bitcoin. Tendermint operates in rounds, where each round has an elected proposer determined in a round-robin fashion based on the stake in the network. Once participating nodes in the network reach a consensus regarding the proposed block, they commit the block to the blockchain at the current height, advance the height by one and reset the rounds to 0 [8][7].

Each round consists of three stages:

- **Proposal**: The proposer suggests a block to be added to the blockchain.
- **Prevote**: Nodes vote on the validity of the proposed block.
- **Precommit**: Nodes confirm their agreement on the block to finalize the decision.

The protocol defines specific message types—`PROPOSAL`, `PREVOTE`, and `PRECOMMIT`—to facilitate communication during these stages. Figure 1 illustrates the Tendermint consensus process.

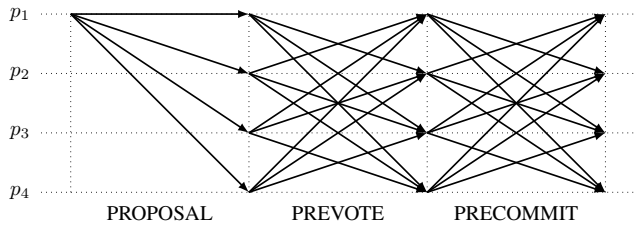

Figure 1: Tendermint Consensus Process

The protocol assumes $n > 3f$, where n is the total voting power of all nodes and f is the aggregate voting power of faulty nodes. To ensure fault tolerance, the total voting power of faulty nodes must be less than one-third of the total. Tendermint also relies on synchronous message delivery and a reliable gossip protocol, which guarantees that if a correct replica *p* sends some message *m*, all correct replicas will eventually receive it [7]. This communication property is crucial for the BFT property of this algorithm as without it it cannot guarantee that the necessary aggregate stake of messages received will be over $2f + 1$.

Testing Approaches

This research utilized the ByzzBench framework to implement and test the Tendermint protocol. ByzzBench is designed to facilitate the evaluation of BFT protocols by supporting multiple testing approaches. For this study, two primary approaches were employed:

- **Baseline Testing**: This approach arbitrarily injects faults into the protocol's execution. Messages are either mutated using predefined mutator structures or dropped entirely. This diverges from Winter et al. [4], as we wanted to be able to compare the performance of different mutations in the baseline approach. Still, no structured fault injection is applied. Baseline testing provides a straightforward method for identifying general issues in protocol behaviour.
- **ByzzFuzz**: This approach introduces structured, round based randomized faults, including network and process failures. As Tendermint terminology also utilises

rounds to track the process of consensus, from now on the rounds in the context of ByzzFuzz will be referred to as **Message-Rounds**. An example of progress of those rounds in regards to Tendermint can be found in Appendix A. ByzzFuzz simulates real-world conditions such as network partitions and process crashes to uncover nuanced violations in safety and liveness properties using structure aware mutations [4]. To compare their performance we have decided to use the the ones originally defined for ByzzFuzz in the baseline testing method too.

ByzzBench supports these testing methods for the same implementation, eliminating the need to modify the protocol to suit each approach. This feature streamlines the testing process and allows for consistent comparisons across methodologies.

3 Contributions

This section presents the main contributions of this research, starting with a review of existing literature on testing BFT protocols in general and Tendermint specifically. It then describes the implementation of the Tendermint protocol within the ByzzBench framework, detailing the modifications and design choices made to enable effective testing.

3.1 Literature Review

The testing of BFT protocols is critical for ensuring correctness and robustness in distributed systems. This section reviews key studies on methodologies and tools for analysing and testing BFT protocols and explores Tendermint related studies.

Testing BFT Protocols

Many testing approaches have been developed to analyse and validate consensus protocols, each with a distinct focus. Crucially, these methods target various aspects which often differ. While some tools emphasize deterministic testing or state-aware fuzzing, others specialize in generating Byzantine attack scenarios or behaviour-divergent models.

Testing approaches that do not focus on Byzantine-specific faults, such as Netrix [9] and LOKI. [10], provide valuable tools for analysing protocol correctness under certain fault conditions. Netrix is a domain-specific language for deterministic testing of consensus implementations, and LOKI is a state-aware fuzzing framework that dynamically models protocol states to generate targeted test cases. For example, LOKI successfully uncovered 20 vulnerabilities, primarily related to memory and consensus logic bugs.

Other tools, such as Twins [11], Tyr [12], and ByzzFuzz [4], focus explicitly on detecting Byzantine-specific faults, though they employ different strategies to emulate adversarial behaviours. The Twins framework, systematically generates Byzantine attack scenarios by duplicating node behaviours to emulate malicious actions. Tyr is a behaviour-divergent model for identifying consensus failure bugs in blockchain systems. It identified 20 previously unknown vulnerabilities across multiple platforms. And lastly Winter et al. [4] introduce ByzzFuzz, a randomized testing tool that detects liveness and safety violations in BFT protocols, including Tendermint. ByzzFuzz employs structured fault injection to uncover subtle vulnerabilities.

Beyond tools, some studies highlight specific vulnerabilities in BFT protocols without implementing testing frameworks. For instance, Berger et al. [13] analyze a 20-year-old optimization in PBFT's [14] read-only request handling. They demonstrate an attack where a Byzantine leader can isolate replicas, breaking the liveness guarantees of the protocol.

Tendermint and Related Work

Tendermint, a BFT consensus protocol, has been extensively analysed for its theoretical correctness and fault tolerance. Amoussou-Guenou et al. [15] rigorously studied Tendermint under synchronous and eventually synchronous models, proving its correctness while highlighting critical mechanisms such as proposer replacement for ensuring fault tolerance under adversarial conditions.

Other studies have also contributed to formal proofs of Tendermint's safety and liveness properties. Kwon's original work [8] established its foundational correctness under partial synchrony. Buchman [16] formalized Tendermint's design and proved its guarantees for safety and liveness in practical deployments. Collectively, these works demonstrate that Tendermint adheres to the theoretical principles of BFT consensus, ensuring consistent and fault-tolerant behaviour.

Conclusion

Numerous testing approaches, such as Netrix, LOKI, and Twins, have been developed to detect violations in BFT protocols, each with unique strengths. ByzzFuzz in particular introduces a different testing approach which focuses on randomly injecting faults into the execution of the protocol to find faults. Its use of precise mutation strategies, like small-scope mutations, makes it especially effective at targeting state-specific faults.

Tendermint, as a widely adopted BFT protocol, has been extensively analysed through both theoretical and practical evaluations. Theoretical works have established its foundational guarantees of safety and liveness under partial synchrony [8][16], while testing efforts, including Netrix and ByzzFuzz, have uncovered protocol deviations and implementation-specific bugs in Tendermint.

3.2 Implementation of the Tendermint Protocol in ByzzBench

Replica Initiation

The implementation was written in the ByzzBench framework. The most important parts are included in the report, however for conciseness, certain logic is omitted or simplified. The main component which had been implemented was the *TendermintReplica* which handled the majority of the logic which was necessary for the protocol to function properly.

This implementation makes use of the voting power structure. As previously mentioned, Tendermint utilizes the PoS mechanism, in which participating nodes may have a different stake in the network. This implies that they are supposed

to be more likely to be elected a proposer for a round and given a chance to propose a block to be committed, if they have a higher stake in the network. In order to implement this functionality, the Tendermint Replica stores the stake of all the replicas in the *votingPowers* map, which is only used to determine the order of proposers, however in this implementation all of the replicas have the same voting power when it comes to reaching a consensus, so each replicas vote is worth the same voting power. The initialization of the Tendermint Replica can be found in the Algorithm 1.

Algorithm 1 Tendermint Replica initialization

function TENDERMINTREPLICA(nodeId, nodeIds, scenario, tolerance, votingPowers)
$height \leftarrow 0$
$round \leftarrow 0$
$step \leftarrow$ PROPOSE
$lockedValue \leftarrow$ NULL
$lockedRound \leftarrow -1$
$validValue \leftarrow$ NULL
$validRound \leftarrow -1$
$messageLog \leftarrow$ New MessageLog()
$tolerance \leftarrow 1$
$TIMEOUT \leftarrow 10$ seconds
$votingPowers \leftarrow$ votingPowers
$ByzzFuzzRounds \leftarrow 0$
end function

Handling requests from the client

To further conform the implementation to ByzzBench, the protocol had to be modified to deal with the client requests. The original implementation, allows replicas to come up with their own values. However, ByzzBench makes use of clients, which send requests to replicas for them to decide whether to commit the block to the chain or not.

Because of this, the changes to the protocol were made that upon receiving a request from the client, a replica will buffer and share this request to all of the other replicas, which subsequently also buffer it for future use. Then when a replica is set to propose a value to other replicas it fetches a value from the buffered requests. A request is then removed from the buffer upon being committed by a replica.

Messages

In order to implement Tendermint in ByzzBench, certain changes had to be made to the already defined message types in Tendermint and can be found in Table 1. The *id* field specifies the sender of the `PROPOSAL`, which is crucial in verification whether the received `PROPOSAL` is coming from a valid proposer. The *h* and *r* correspond to the $height$ and $round$ at which the message had been sent accordingly. The *vr* stands for $validRound$ and the *v* stands for the $block$ which replicas are attempting to commit to the blockchain.

The logic which allows for utilisation of ByzzFuzz, is based heavily on those messages and can be found in Appendix A. This directly corresponds to how Winter et al. [4] used ByzzFuzz.

Messages
`PROPOSAL`(*id*, *h*, *r*, *vr*, *v*)
`PREVOTE`(*id*, *h*, *r*, *v*)
`PRECOMMIT`(*id*, *h*, *r*, *v*)
`GOSSIP`(*id*, *message*)

Table 1: Overview of messages

Message Log

The MessageLog, found in Algorithm 2, allows for storing all received messages for a replica. It is a crucial part of the implementation as upon receiving a message, it attempts to add it to the appropriate SortedMap. The choice to use a SortedMap, where the key is the block from the message and the value is a SortedSet provided guarantees that a message would only be considered once in the process of reaching a consensus. Additionally it simplified the necessary logic for fetching all the messages for a certain block and ensured a deterministic approach.

Algorithm 2 MessageLog Class

Attributes:
$tolerance \leftarrow$ (long)
$votingPower \leftarrow$ SortedMap<String, Integer>
$messages \leftarrow$ SortedSet<GenericMessage>
$prevotes \leftarrow$ SortedMap<Block, List<PrevoteMessage>>
$precommits \leftarrow$ SortedMap<Block, List<PrecommitMessage>>
$proposals \leftarrow$ SortedMap<Block, List<ProposalMessage>>
$requests \leftarrow$ Set<RequestMessage>

Gossip communication

In order to fulfil the Tendermints assumption of a reliable gossip communication protocol, a new message type was introduced, namely the `GOSSIP` message. It contains the *id* of the sender and the message it is relaying. In order to ensure reliable delivery, ByzzFuzz was configured to not mutate nor drop gossip messages. The gossip protocol specified earlier has been implemented in the manner demonstrated in the example below.

Assume a replica *p1* sends a message `M` to a replica *p2* and *p3*. Upon receiving the message `M` the *p2* and *p3* replicas will in turn relay this message to all the other replicas in a `GOSSIP` message in order to ensure that even if any replica missed out on the message, it will eventually receive it. Now the *p2* replica upon receiving the `GOSSIP` from *p3* will not gossip it further as it already has received that message before, and sent out a `GOSSIP` itself.

Handling of messages

In Buchman et al. [7] a set of rules is specified which upon being fulfilled execute some specified logic. In order for a more maintainable implementation we have opted into creating a more modular solution. A replica upon receiving a message,

calls a method which is responsible for handling this specific message type.

Algorithm 3 Handle Proposal

```
function HANDLEPROPOSAL(proposalMessage)
    uponRules ← [false, false, false, false]
    if FULFILLPROPOSALRULE0(proposalMessage) then
        uponRules[0] ← true
    end if
    if FULFILLPROPOSALRULE1(proposalMessage) then
        uponRules[1] ← true
    end if
    if FULFILLPROPOSALRULE2(proposalMessage) then
        uponRules[2] ← true
    end if
    if FULFILLPROPOSALRULE3(proposalMessage) then
        uponRules[3] ← true
    end if
    PROPOSALRANDOMORDEREXECUTE(uponRules,
proposalMessage)
end function
```

As an example we will consider the *handleProposal* method, found in Algorithm 3. There are four rules which take into consideration a `PROPOSAL` message, hence if a rule is satisfied, based on the messages it has received in the past, it marks that the corresponding logic should be executed. Buchman et al. [7] specifies that satisfied rules should be executed in a random order. For this reason, the order is shuffled using a Random object. We used the same seed to create this object each time and the same sequence of method calls is made for it, hence each execution of the schedule will be identical [17]. This way the concern of nondeterminism is handled, and the execution order is repeatable. This is crucial for testing purposes, as in the case that nondeterminism would be introduced, a schedule that was previously marked as one with either a liveness or agreement violation, upon materializing could go down a different order of execution and produce a different outcome.

4 Experimental Setup and Results

This section outlines the experimental methodologies and configurations used to evaluate the Tendermint implementation within the ByzzBench framework. It describes the mutation strategies, fault scenarios, and testing parameters employed by ByzzFuzz and baseline methods. Results from these tests are presented and used to answer the previously established research questions.

4.1 Mutators

In order to use ByzzFuzz to test the implementation, certain structure-aware mutations had to be implemented, and their overview can be found in Table 2. The parameters in the messages correspond to the ones described in the Table 1. There are two types of mutations, namely small-scope (SS) and any-scope (AS). Small-scope mutations are mutations which change the mutated value by one, i.e. from 1 to 2 or from 1 to 0. Any-scope mutations in turn modify the field arbitrarily.

Message	Mutation
`PROPOSAL`(*id*, *h*, *r*, *vr*, *v*)	`PROPOSAL`(*id*, *h'*, *r*, *vr*, *v*)
	`PROPOSAL`(*id*, *h*, *r'*, *vr*, *v*)
`PREVOTE`(*id*, *h*, *r*, *v*)	`PREVOTE`(*id*, *h'*, *r*, *v*)
	`PREVOTE`(*id*, *h*, *r'*, *v*)
`PRECOMMIT`(*id*, *h*, *r*, *v*)	`PRECOMMIT`(*id*, *h'*, *r*, *v*)
	`PRECOMMIT`(*id*, *h*, *r'*, *v*)

Table 2: State aware mutations

Potential Vulnerabilities

In Winter et al. [4] there is a potential violation found in their implementation of the Tendermint protocol. Tendermint's reliance on the gossip communication introduces vulnerabilities in scenarios where its assumptions are violated. For example in a faulty gossip communication, messages may not be delivered to all nodes when under network partitions. This potentially leads to termination violations. In order to recreate the known violation using ByzzFuzz within the ByzzBench framework, we have ensured that a network fault would be injected during the `PRECOMMIT` **Message-Round**, directly following the reported potential violation. The logic it uses is shown in Figure 2. Specifically, this partition only allows replicas *p1* and *p2* to send each other `PRECOMMIT` messages in that **Message-Round**.

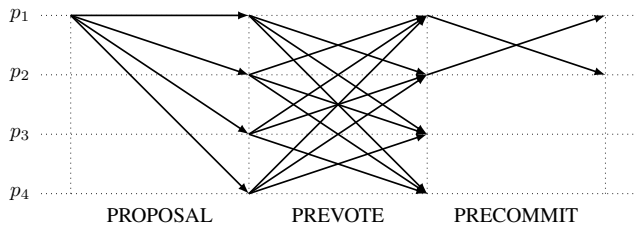

Figure 2: Tendermint Consensus Process

4.2 Overall Setup

We have ran all the testing approaches with a total of 4 replicas, and at most one faulty one. This satisfied the assumption of Tendermint of $n > 3f$ given that each replicas had an equal stake of 1 in the network.

In the testing process, ByzzBench allows to define multiple parameters either for testing or termination purposes, however some of them apply to multiple testing approaches and will be described here. The parameters specific to one approach are described in the section relevant to that approach. The termination variables always apply and are as follows: *minEvents*, *minRounds* and *samplingFrequency*, which for all of the simulated schedules were set to 1000, 10 and 1 respectively. The *minEvents* and *minRounds* are the amount of events and rounds respectively which had to have passed in the schedule for the simulation to terminate. The *samplingFrequency* specifies how often the simulator should check whether the conditions are satisfied, for the scheduler to be able to determine the next action *deliverMessageWeight*

and *deliverClientRequestWeight* were set to 10000 each. This was due to a bug found in the ByzzBench framework, which with lower probabilities allowed the scheduler to execute a timeout out of order, creating violations that were not due to the implementation of the Tendermint protocol.

4.3 Baseline Testing

To evaluate the performance of the baselines testing method, we have run the random event scheduler with different parameters. The *dropMessageWeight* and *mutateMessageWeight* are denoted by **D** and **M** respectively. The weights determine the probability of a message being either dropped or mutated, with the higher probabilities making it more likely for the fault happening. The Table 3 presents the faults found by the baseline testing method with different parameters, distinguishing them into liveness and agreement violations. Those in turn are divided into small-scope and any-scope, indicating what mutation type found those violations. The results show an increase in violations detected that grows proportionally to the *dropMessageWeight*.

D	M	Liveness		Agreement		Total
		SS	AS	SS	AS	
0	0	0	0	0	0	2000
0	1000	0	0	0	0	2000
0	2000	1	0	0	0	2000
1000	0	2	2	0	0	2000
1000	1000	7	2	0	0	2000
1000	2000	8	10	0	0	2000
2000	0	31	34	0	0	2000
2000	1000	35	37	0	0	2000
2000	2000	34	30	0	0	2000

Table 3: Faults found by the baseline testing approach

4.4 ByzzFuzz

To compare the performance of those two testing approaches, we ran the ByzzFuzz testing approach with multiple configurations found in Tables 4, 5 and 6(found in Appendix C). Those investigated how ByzzFuzz would perform some configuration of the assumptions of reliable gossip communication and synchronous message delivery. All of them distinguish two types of faults injected, where **P** stands for process faults and **N** for network faults. The baseline setup is a setup in which no faults are injected. Furthermore, all simulations were run with *numRoundsWithFaults* set to 8, which should schedule faults up to the eight **Message-Round** in the scenario. This allowed scenarios to have up to two rounds with no faults injected. This was to directly adhere to the way ByzzFuzz was run by Winter et al. [4]. The tables present the faults found by ByzzFuzz in different configurations, distinguishing them into liveness and agreement violations. Those in turn are divided into small-scope and any-scope, indicating what mutation type found those violations.

Faults	Liveness		Agreement		Total
$baseline$	0		0		2000
	SS	AS	SS	AS	
$\mathbf{P}=0,\mathbf{N}=1$	0	0	0	0	2000
$\mathbf{P}=0,\mathbf{N}=2$	0	0	0	0	2000
$\mathbf{P}=1,\mathbf{N}=0$	1	0	0	0	2000
$\mathbf{P}=1,\mathbf{N}=1$	1	0	0	0	2000
$\mathbf{P}=1,\mathbf{N}=2$	1	2	0	0	2000
$\mathbf{P}=2,\mathbf{N}=0$	4	0	0	0	2000
$\mathbf{P}=2,\mathbf{N}=1$	8	1	0	0	2000
$\mathbf{P}=2,\mathbf{N}=2$	4	2	0	0	2000

Table 4: Faults found by ByzzFuzz on the implementation fulfilling both synchronous message delivery and gossip protocol assumptions

Faults	Liveness		Agreement		Total
	SS	AS	SS	AS	
$\mathbf{P}=0,\mathbf{N}=0$	0	0	0	0	2000
$\mathbf{P}=0,\mathbf{N}=1$	441	441	0	0	2000
$\mathbf{P}=0,\mathbf{N}=2$	669	701	0	0	2000
$\mathbf{P}=1,\mathbf{N}=0$	1	0	0	0	2000
$\mathbf{P}=1,\mathbf{N}=1$	397	420	0	0	2000
$\mathbf{P}=1,\mathbf{N}=2$	657	666	0	0	2000
$\mathbf{P}=2,\mathbf{N}=0$	5	0	0	0	2000
$\mathbf{P}=2,\mathbf{N}=1$	381	421	0	0	2000
$\mathbf{P}=2,\mathbf{N}=2$	644	685	0	0	2000

Table 5: Faults found by ByzzFuzz on the implementation fulfilling the synchronous delivery of messages and using unreliable gossip protocol

4.5 Discussion On Results

In this section, we analyse the results of the tests conducted to answer the research questions (RQ1, RQ2 and RQ3). The goal is to provide context for the findings, draw comparisons, and explore the implications for the testing of BFT protocols.

Can ByzzFuzz find any bugs in the implementation of Tendermint?

ByzzFuzz successfully identified several bugs in the implementation of Tendermint, which were deviations from its intended behaviour. These bugs highlight critical issues related to the protocol's handling of round advancement.

One critical bug was identified in the rule responsible for advancing a round upon receiving $f + 1$ messages with a round number higher than the current round stored in a replica. The issue arose from a divergence between the intended functionality and the actual implementation. Specifically, the buggy implementation mistakenly advanced the round for any $f + 1$ messages where the round number differed from the replica's current round, instead of ensuring the new round was strictly greater. This discrepancy is highlighted in Algorithm 4, which shows the faulty logic.

For example, consider a mutation that changes the round number in a message from 0 to -1. If a replica receives $2 =$

$f + 1$ (with $f = 1$) such mutated messages, i.e. PROPOSAL and PREVOTE messages from the same faulty node, the buggy protocol incorrectly advanced the replica to round -1. This round should never have appeared within the protocol's design, resulting in replicas entering an invalid state. Once in this state, replicas are unable to recover, ultimately violating the protocol's liveness guarantees. This demonstrates a critical flaw in handling invalid state transitions when processing faulty or malicious inputs.

Algorithm 4 Buggy MoveOn Rule

upon $f + 1\langle *, \text{height}, \text{round}, *, * \rangle$ **with** $\text{round} \neq \text{round}_p$

The correct implementation of the rule, as shown in Algorithm 5 taken from Buchman et al. [7], ensures that a replica only advances to a new round when it receives $f + 1$ messages with a round number strictly greater than the current round stored in the replica. This guarantees that replicas do not transition to invalid or undefined states.

Algorithm 5 Correct MoveOn Rule

upon $f + 1\langle *, \text{height}, \text{round}, *, * \rangle$ **with** $\text{round} > \text{round}_p$

After identifying and correcting the bug, the faulty schedules were retested, and no violations were observed. This confirmed that the bug in Algorithm 4 was the root cause of the detected liveness violations.

Another bug that ByzzFuzz successfully identified also involved this method, however this time it was the way it was calling it. One usage simply was giving the wrong height for the comparison. Making it in certain situations unable to correctly move on to a higher round.

This leads us to conclusion that ByzzFuzz successfully found issues in the implementation of the Tendermint protocol. While other violations are still reported we were unable to determine their root cause. Still, it can not be said that those would be present in a bug-free implementation that may be in use on a functioning system employing the Tendermint protocol.

How does the bug detection performance of ByzzFuzz compare to a baseline testing method that arbitrarily injects network and process faults?

A direct comparison of the baseline testing method and ByzzFuzz reveals significant differences in their bug detection capabilities. At first glance, the baseline testing method appears to perform better. However, closer examination reveals a critical flaw: the baseline testing method directly violates the Tendermint protocol's assumption of a reliable gossip communication mechanism. Specifically, while ByzzFuzz restricts the dropping of messages to only PROPOSAL, PREVOTE, and PRECOMMIT, the baseline method also drops GOSSIP messages, which breaches the protocol's core assumptions.

To investigate the impact of improperly functioning gossip communication in the presence of structured faults, ByzzFuzz was tested in a configuration that allowed gossip messages to be dropped (Table 5). When comparing only the configurations that adhere to Tendermint's assumptions, ByzzFuzz consistently detected more violations than the baseline method. This finding suggests that ByzzFuzz is more efficient in uncovering implementation-specific faults.

The disparity between the two methods highlights the advanced fault injection capabilities of ByzzFuzz. While the baseline identified liveness violations too, ByzzFuzz detected a broader range of issues, including agreement violations and did not violate the assumptions of the protocol. ByzzFuzz's ability to simulate network partitions—a critical real-world fault scenario where subsets of replicas are temporarily isolated—enabled it to identify vulnerabilities that the baseline's simplistic, random fault injections failed to uncover.

How do small-scope and any-scope message mutations of ByzzFuzz compare in their performance of bug detection for Tendermint?

The results presented in Table 4, combined with the previously identified bug, indicate that small-scope (SS) message mutations outperform any-scope (AS) mutations in their ability to trigger protocol violations. With only AS mutations, the specific bug allowing replicas to transition to an invalid round was significantly less likely to occur. This is because AS mutations require two independent random mutations to assign the same invalid round value, which, due to the high range of possible values, was never observed.

In contrast, SS mutations incrementally modify values, making it more likely for specific faults, such as invalid round transitions, to be triggered. This advantage demonstrates the precision of SS mutations in targeting protocol vulnerabilities, particularly those that depend on small, state-specific deviations.

Evaluating the resilience of our implementation against known violations

As discussed in section 4.1, the identified violation was further investigated under different configurations. In ByzzFuzz configurations that adhered to Tendermint's assumption of reliable gossip communication,specifically where GOSSIP messages were not dropped, the violation was never observed after running the scenario more than two thousand times. In such cases, other replicas reliably relayed dropped messages during partitions, ensuring eventual correctness.

However, in configurations where the gossip protocol was allowed to violate its assumptions, the implementation consistently failed to handle these faults across all scenarios. Without the ability to rely on the gossip protocol to retransmit missing messages, the system could not recover, resulting in persistent violations. This demonstrates the protocol's dependency on reliable message delivery and highlights the critical role of robust gossip mechanisms in ensuring liveness and correctness.

Furthermore, we have created additional network faults that followed the same partitions, but instead of happening only for PRECOMMIT messages, they were extended to be scheduled for PREVOTE messages. The results were identical to those found by running the faults mimicking the known potential violation. Therefore, we report another potential violation stemming from the same root cause, as this violation

is only found when violating the network assumptions of the Tendermint protocol, like the one found in Winter et al. [4].

General Insights

Apart from answering the research questions, certain observations were made. Despite the identified bugs, this Tendermint implementation demonstrated strong resilience to network faults. Even under configurations with multiple network faults injected, the protocol maintained its ability to recover and continue processing. This resilience stems from its design, which heavily relies on a robust gossip communication protocol and synchronous message delivery to propagate critical information reliably. We have tested the implementation further by only fulfilling one of the assumptions of the Tendermint protocol in Tables 5 and 6. Those results only further highlighted the significance of those assumptions.

5 Responsible Research

This section discusses the ethical considerations, reproducibility, and broader impact of the research conducted in this paper. It highlights the adherence to responsible research principles, the documentation of configurations for reproducibility, and the limitations of the study. The section also emphasizes the role of this work in advancing the reliability of distributed systems and promoting sustainable practices in consensus protocol design.

5.1 Ethical Considerations

The research conducted in this project adheres to the principles of responsible and ethical research. The implementation and testing of the Tendermint protocol in ByzzBench focused only on evaluating the reliability and correctness of the protocol under simulated conditions. No real-world systems or production environments were involved in the writing of this paper, ensuring no disruption to operational services or users.

5.2 Reproducibility

To promote reproducibility, all experiments were conducted using the ByzzBench framework, which provides a standardized environment for testing BFT consensus protocols. The specific configurations and parameters used during the experiments have been explicitly documented in the "Experimental Setup and Results" section.

The implementation of the Tendermint protocol, along with the test schedules and mutation configurations, will be available as part of the supplementary materials accompanying this paper. This ensures that other researchers can replicate and verify the findings presented in this work.

5.3 Limitations and Responsible Use

While the research highlights potential vulnerabilities in Tendermint's implementation, the intention is not to exploit these weaknesses but to emphasize the importance of robust testing frameworks like ByzzBench. The results should be interpreted in the context of improving protocol reliability and advancing the field of distributed systems research.

Researchers and practitioners are encouraged to use the insights from this work to strengthen the implementation and testing of BFT consensus protocols rather than identifying opportunities for malicious exploitation.

5.4 Broader Impact

The increasing reliance on blockchain and distributed systems underscores the importance of ensuring their correctness and reliability. By promoting rigorous testing practices, this research contributes to building trust in these technologies, which are critical to modern financial, social, and operational systems. At the same time, the energy-efficient mechanisms explored in this paper, such as Proof-of-Stake, align with global efforts to reduce the environmental impact of computational systems.

6 Discussion

In this section, we compare the results obtained by us to the results from previous relevant works. Our results align with findings from Winter et al. [4], who demonstrated the utility of structured fault injection for detecting liveness and agreement violations in Tendermint. This work expanded upon their findings by identifying an additional potential termination violation, which was an extension of a previously reported issue. The results indicated flaws in the implementation however they did not allow us to find any issues that were previously not identified. This aligns with findings from other works, such as Amoussou-Guenou et al. [15], who provided formal proofs of Tendermint's safety and liveness under synchronous and eventually synchronous models, confirming its correctness when its core assumptions are met. Additionally, Buchman et al. [7] and Kwon [8] established Tendermint's safety and liveness guarantees under its core assumptions.

These validations support our conclusion that while Tendermint's theoretical design is sound, its implementation is susceptible to flaws when assumptions about network reliability or synchronous communication are violated. This emphasizes the importance of rigorous testing and structured fault injection to ensure the reliability of real-world deployments.

7 Conclusions

This study utilized ByzzFuzz within the ByzzBench framework to evaluate the implementation of the Tendermint consensus protocol, focusing on its bug detection capabilities, comparative performance against baseline testing, and the impact of different mutation strategies.

ByzzFuzz proved effective in uncovering critical faults in our Tendermint implementation, namely liveness violations. Notably, it identified a significant bug related to incorrect round advancement caused by invalid state transitions. The study also confirmed Tendermint's reliance on assumptions of synchronous message delivery and a reliable gossip communication protocol, with violations of these assumptions resulting in persistent faults.

In comparison of ByzzFuzz with baseline testing, ByzzFuzz demonstrated better performance in detecting implementation-specific faults while maintaining the protocol's core assumptions. The baseline testing method, although effective at identifying a broader range of faults, often violated the protocol's design principles, leading to misleading results. This underscores the importance of structured fault injection in testing BFT protocols.

The comparison of small-scope (SS) and any-scope (AS) mutations revealed that SS mutations are more effective at uncovering faults tied to state-specific transitions, such as invalid round advancements. By incrementally modifying values, SS mutations targeted subtle vulnerabilities more effectively than AS mutations, which struggled with the high variability in arbitrary value selection.

This research also highlighted Tendermint's resilience to transient network faults when its core assumptions were upheld. However, when these assumptions were relaxed, such as allowing the gossip protocol to fail or introducing asynchronous message delivery, the protocol struggled to maintain the liveness property. These findings emphasize the critical role of both synchronous communication and robust message propagation in ensuring Tendermint's reliability.

In conclusion, this study demonstrates the value of ByzzFuzz as a structured testing tool for identifying nuanced vulnerabilities in BFT protocols. By enabling targeted fault injections and precise analysis, ByzzFuzz complements other testing methods to provide a comprehensive evaluation of consensus protocol implementations.

8 Future Work and Improvements

While this research provides valuable insights, there remain several areas for improvement regarding both the implementation of Tendermint and the testing approaches utilized in this paper.

For the Tendermint implementation, one possible improvement involves the incorporation of a power voting mechanism. Currently, all tests were conducted in a democratic scenario where each replica held equal voting power. A more realistic approach would involve assigning voting power based on the stake of each replica. This adjustment could help investigate the protocol's behavior in scenarios where validators have unequal influence, providing insights into vulnerabilities that may arise when a small number of nodes hold significant power.

Improvements to the testing approaches used in this research could also enhance the outcomes. The baseline testing approach, as described in Section 4.5, allows the dropping of all messages, which may violate the protocol's assumptions about reliable gossip communication. Restricting the baseline so that critical messages, such as `PROPOSAL`, `PREVOTE`, and `PRECOMMIT`, are not dropped could provide results that better reflect real-world conditions. Additionally, the structure-aware mutation strategy currently modifies only the *height* and *round* fields in messages. Expanding the scope of mutations to include other fields, such as the *validRound* could help uncover new vulnerabilities in the implementation and lead to a more comprehensive evaluation of the protocol.

A Mapping Tendermint Rounds to ByzzFuzz Rounds

In the Tendermint consensus protocol, replicas attempt to commit a block for each height through multiple rounds. Within a height, each round has a designated proposer, and replicas collectively proceed through three phases—Proposal, Prevote, and Precommit—before advancing to the next round or committing a block. If a round fails to commit a block, the protocol advances to the next round at the same height with a new proposer. Once a block is successfully committed, the height increments, and the process repeats.

We created a mapping from the combination of height (h) and round (r), to utilise ByzzFuzz's round based approach. For example, in Tendermint, the phases of *h=0, r=0* would map to:

- `PROPOSAL`(*h0r0*) = 1
- `PREVOTE`(*h0r0*) = 2
- `PRECOMMIT`(*h0r0*) = 3

If this round fails, the replicas move to *h=0, r=1*:

- `PROPOSAL`(*h0r1*) = 4
- `PREVOTE`(*h0r1*) = 5
- `PRECOMMIT`(*h0r1*) = 6

Upon successfully committing a block in *h=0, r=1*, the height increments (*h=1*), and the process starts over:

- `PROPOSAL`(*h1r0*) = 7
- `PREVOTE`(*h1r0*) = 8
- `PRECOMMIT`(*h1r0*) = 9

This mapping illustrates how ByzzFuzz models Tendermint's consensus process. Each height-round combination corresponds to a unique sequence of `PROPOSAL`, `PREVOTE`, and `PRECOMMIT` phases, allowing structured analysis of the protocol's fault-handling capabilities.

B Use of LLMs

During the research project, ChatGPT was used for checking grammatical correctness and ensuring that the text is clear. The prompts used included:

- "What would you suggest to you improve the grammar? [input]"
- "Can you summarize this paragraph and tell me about it? [input]" This prompt was used to determine whether the main idea of the the text written was clear. If the response summarized the paragraph incorrectly, it would indicate that we have to write it better.
- How do you format a table in overleaf?
- How would you use passive voice in that?[input]

Additionally Grammarly and Writefull were used to ensure correct spelling and grammar.

C Additional Results

This section presents additional results from the ByzzFuzz evaluation of the Tendermint implementation under asynchronous message delivery and a reliable gossip communication assumption.

Faults	Liveness		Agreement		Total
	SS	AS	SS	AS	
$\mathbf{P} = 0, \mathbf{N} = 0$	92	83	0	0	2000
$\mathbf{P} = 0, \mathbf{N} = 1$	81	98	0	0	2000
$\mathbf{P} = 0, \mathbf{N} = 2$	93	87	0	0	2000
$\mathbf{P} = 1, \mathbf{N} = 0$	90	79	0	0	2000
$\mathbf{P} = 1, \mathbf{N} = 1$	90	67	0	0	2000
$\mathbf{P} = 1, \mathbf{N} = 2$	73	100	0	0	2000
$\mathbf{P} = 2, \mathbf{N} = 0$	65	90	0	0	2000
$\mathbf{P} = 2, \mathbf{N} = 1$	66	77	0	0	2000
$\mathbf{P} = 2, \mathbf{N} = 2$	83	78	0	0	2000

Table 6: Faults found by ByzzFuzz on the implementation using asynchronous message delivery and fulfilling the reliable gossip communication assumption.

References

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

[2] E. Kapengut and B. Mizrach, "An event study of the ethereum transition to proof-of-stake," *Commodities*, vol. 2, no. 2, pp. 96–110, 2023.

[3] C. Staff, "This week in crypto: Binance experiences a $100 million cross-chain hack," *CoinJournal*, 2022. Accessed: 2025-01-07.

[4] L. N. Winter, F. Buse, D. de Graaf, K. von Gleissenthall, and B. Kulahcioglu Ozkan, "Randomized testing of byzantine fault tolerant algorithms," *Proc. ACM Program. Lang.*, vol. 7, Apr. 2023.

[5] K. John, T. J. Rivera, and F. Saleh, "Economic implications of scaling blockchains: Why the consensus protocol matters," *Available at SSRN 3750467*, 2020.

[6] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Berlin, Heidelberg: Springer-Verlag Berlin Heidelberg, 2nd ed., 2011.

[7] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on BFT consensus," *CoRR*, vol. abs/1807.04938, 2018.

[8] J. Kwon, "Tendermint: Consensus without mining." Draft v.0.6, 2014.

[9] C. Dragoi, C. Enea, S. Nagendra, and M. Srivas, "A domain-specific language for testing consensus implementations," in *arXiv preprint*, vol. arXiv:2303.05893v2, 2023.

[10] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, "Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols," in *Network and Distributed System Security Symposium (NDSS)*, 2023.

[11] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," in *42nd Conference on Very Important Topics (CVIT)*, pp. 23:1–23:29, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[12] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *2023 IEEE Symposium on Security and Privacy (SP)*, pp. 2517–2532, IEEE, 2023.

[13] C. Berger, H. P. Reiser, and A. Bessani, "Making reads in bft state machine replication fast, linearizable, and live," in *40th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, IEEE, 2021. Preprint version available.

[14] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (USA), p. 173–186, USENIX Association, 1999.

[15] Y. Amoussou-Guenou, A. D. Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Dissecting tendermint," *arXiv preprint*, vol. arXiv:1809.09858v3, 2019.

[16] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Master's thesis, University of Guelph, 2016.

[17] Oracle Corporation, *Oracle Java 8 API Documentation: java.util.Random*, 2014. Accessed: 2025-01-11.