Testing Zyzzyva

An evaluation and comparison of Byzantine Fault Tolerant algorithm testing strategies

Ishan Singh Pahwa[1]

Supervisor(s): Dr. Burcu Kulahcioglu Ozkan [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 27, 2025

Name of the student: Ishan Singh Pahwa
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Külahçıoğlu Özkan, João Miguel Louro Neto, Dr. Jérémie Decouchant

Abstract

Testing Byzantine Fault Tolerant (BFT) algorithms is crucial in uncovering potential liveness and safety violations for distributed systems. This paper focuses on testing Zyzzyva with ByzzFuzz and Twins and evaluating their performance with each other and a baseline testing strategy. We also investigate if ByzzFuzz can uncover faults in Zyzzyva, and how small-scope mutations compare to any-scope mutations. We also discuss limitations with ByzzFuzz when it comes to testing BFT protocols. We find that ByzzFuzz is currently unable to find known safety violations in Zyzzyva, but can find injected violations and Twins does not find violations given our small sample size.

1 Introduction

Byzantine fault tolerant (BFT) algorithms are important when it comes to implementing distributed systems as they attempt to reach a consensus between different nodes in the network, some of which may be faulty or act maliciously against the system [5]. Many critical distributed systems therefore rely on BFT algorithms to provide guarantees about the systems' behaviours, including systems such as cryptocurrencies, blockchains and cloud computing. Achieving correct implementations of Byzantine fault-tolerant systems is challenging and implementations have been found to violate at least one of the two guarantees that BFT algorithms try to solve, namely liveness and safety, such as the ones found by Winter et al. in their 2023 paper on the subject [9].

Zyzzyva [4] is a BFT algorithm that innovatively introduced speculation, a technique where nodes can execute operations optimistically by adopting the order proposed by the primary. This technique has been shown to improve performance by increasing throughput and decreasing latency as it reduces replica overheads, especially cryptographic hashes, to their theoretical minimum. Zyzzyva however was shown to have a safety violations by Abraham et al. [1] in 2017 during its view change sub-protocol.

ByzzFuzz introduces a novel solution for the testing of BFT algorithms using small-scope mutations, a testing strategy that mutates the content of protocol messages by applying small changes to their values, for example, incrementing or decrementing the proposed sequence number of a request, and combining it with existing testing strategies found in literature [9]. This approach to testing demonstrates that ByzzFuzz has been shown to uncover previously unknown bugs in production BFT systems [9]. Despite its promise, ByzzFuzz has only been applied to a limited number of algorithms, leaving its broader applicability and performance in testing other BFT algorithms largely unexplored.

In this paper, we focus on testing Zyzzyva using ByzzBench, a BFT testing suite that integrates ByzzFuzz, a baseline fault injection testing strategy as well as Twins, a testing strategy that duplicates nodes to have the same identity which introduces byzantine faults as the system now has two nodes with the same credentials (for example id and private cryptographic keys) but different behaviours [2]. Our aim with this research is to uncover bugs in Zyzzyva as well as study the bug detection performance of ByzzFuzz against the baseline testing strategy and Twins as well as the efficacy of small-scope mutations compared to any-scope mutations.

Therefore, the main questions we aim to answer with this research are:

RQ1 Can ByzzFuzz implemented in ByzzBench find faults in the safety and liveness guarantees that Zyzzyva claims to provide?

RQ2 What is the difference in performance between ByzzFuzz, a baseline testing method and Twins?

RQ3 How do small-scope and any-scope mutations in ByzzFuzz compare in the performance of bug detection?

2 Background

2.1 Zyzzyva

Zyzzyva, introduced by Kotla et al. in 2007 [4] is a leader-based groundbreaking algorithm that pioneered the use of speculative execution in order to reduce the cost and simplify the design of the system by not having an expensive agreement protocol to establish a concrete order but rather have the replicas accept a primary's ordering and eventually sync up if there are discrepancies in the history of the nodes. Zyzzyva also places considerable responsibility on the clients in order to commit the requests as well as find discrepancies in the primary's orderings of the request.

Zyzzyva consists of three sub-protocols, the Agreement, View Change and Checkpoint protocols:

Agreement sub-protocol

The Agreement sub-protocol orders the request for execution by the replicas, speculatively executes it and returns a response to the client. The client only commits to a response from the replicas if it receives at least $3f + 1$ matching responses (considered the fast track) or at least $2f + 1$ acknowledgements of the reception of a commit certificate (considered the two-phase commit). Since histories may diverge in replicas, the agreement sub-protocol offers a way to fill gaps in the history through the use of fill-hole messages when the replica receives a larger sequence number than the one it was expecting.

View Change sub-protocol

The View Change protocol allows the system to change primaries if a primary is found to be unreliable in its ordering or too slow for the network. Each view change allows the opportunity for the replicas to all start with the same history and sync up through the reconciliation of their histories.

Checkpoint sub-protocol

The Checkpoint protocol truncates the state in each replica to cut down on complexity as well as performance overhead in the view change as the view change sub-protocol uses the system's history to bring the replicas up to sync. The sub-protocol also creates a commit-certificate after receiving enough matching responses with the same history. In our implementation of Zyzzyva, we commit when we receive a commit-certificate, so the checkpoint protocol also acts as one of our times to commit to the log.

The faulty cases

In 2017, Abraham et. al showcased two safety flaws in Zyzzyva's view change algorithm [1]. The first scenario showcases that the protocol's prioritisation of commit-certificates for a request in the current view over the $f + 1$ ORDERED-REQUEST messages during the new primary's computation for the new ordered request history can lead to a safety issue causing conflicting log positions over views. The second scenario shows that choosing the commit certificate with the longest request-log during the view change can lead to safety violations. We shall consider the first case and attempt to recreate it in the paper.

2.2 ByzzFuzz

ByzzFuzz, introduced in 2023 by Levin et al., gains its inspiration for testing BFT algorithms by testing strategies for distributed systems that introduce random faults in the network and as a result, isolate specific nodes. Network faults such as partitioning have been shown to be effective at finding problems in distributed systems [6]. ByzzFuzz uses the same approach of injecting random faults, but in the process of the system, rather than the network, to mimic Byzantine Faults in the system[9]. It introduces the notion of small-scope mutations to protocol messages which mutate fields inside the messages, for example, by incrementing the sequence or view numbers in Zyzzyva's ordered request message. The motivation for this was that such small changes in the message would be closer to what the system expects and therefore, not ignored. They also serve as a method of boundary testing. This is in contrast to the any-scope mutations which have a much larger set of values to mutate the message with and are theoretically, less likely to be accepted by the system.

ByzzFuzz also employs fault-bounded testing, a constraint on the number of network and process faults that are introduced into the system; round-based testing, the notion of introducing network or process faults in all messages sent in a given round rather than individual messages; and finally structure-aware mutations, mutations that corrupt the messages that provide valid protocol messages that deviate from normal protocol execution.

ByzzFuzz has managed to find and reproduce several bugs in BFT protocols, such as a liveness violation in the PBFT protocol and an implementation bug in Ripple, however, due to its relative recency, it hasn't been widely tested on different BFT protocols. Given its promise, we expect it to uncover the previously known safety attacks on Zyzzyva.

2.3 Twins

Twins, introduced in 2020 by Bano et al., offers a practical approach to testing BFT algorithms by cloning a replica with the same ID but the "twinned" replica can act maliciously by contradicting the actions of the original replica. For example, in the case of the safety violation mentioned by Abraham et al., a twin could receive a commit certificate for a given request, but the other doesn't. During the view change, the twin without the commit certificate sends what they believe to be the first message in the system, which is accepted. However, in the following view change, the twin with the commit certificate sends their commit certificate in their view change message and this contradicts the system, forming a safety violation. An important difference to note between Twins and ByzzFuzz is that Twins does not mutate the messages of the system but rather relies on these malicious nodes to introduce Byzantine faults.

Twins managed to find the safety violation proposed by Abraham et al., as well as a liveness attack on FaB, a predecessor to Zyzzyva, also with a fast and two-phase track. Finally in their 2020 paper, they also reproduced a both a liveness and safety violation on Sync HotStuff proposed by Momose et al [8]. Given that it managed to find flaws in several popular BFT algorithms, it would appear that Twins is promising when it comes to uncovering violations in other protocols.

2.4 Baseline testing strategy

Our baseline testing strategy is a network wrapper that randomly creates processs and network faults by applying mutations to messages sent in the network or by dropping messages altogether. It differs from ByzzFuzz in that it isn't aware of rounds and therefore does not create round-aware mutations and partitions. It does, however have a distinction between small-scope and any-scope mutations.

2.5 Other testing strategies

Around the time that the paper in which ByzzFuzz was published, the behaviour-divergent testing model Tyr was also published by Chen et al. [3] in which the aim of the testing strategy was to make the nodes in the system have behaviours that diverge as much as possible to find potential vulnerabilities in blockchain BFT algorithms. Hermes is another fault-injection testing framework introduced in 2013 that injects random network and process faults [7].

3 Implementing Zyzzyva in ByzzBench

In order to evaluate the correctness of the safety and liveness guarantees that Zyzzyva provides as well as to compare performances between the testing strategies, we use ByzzBench, a benchmarking testing suite specifically designed to test BFT algorithms by integrating ByzzFuzz, Twins and a baseline testing strategy that introduces random faults into the process and network. ByzzBench provides the ecosystem required to implement BFT protocols with abstract implementations for clients, replicas, messages and scenarios as well as a transport layer for messages to be exchanged and a commit log. ByzzBench is implemented in Java.

It's important to note that ByzzBench sends messages to a mailbox, meaning that a message, when sent isn't delivered directly to the receiver but rather held in a buffer state. This allows messages that were sent later to be processed by replicas before messages that were sent previously. ByzzBench also has a graphical user interface and logging functionalities to a console in order for the user to interact with the program. Furthermore, while mutating messages using the schedulers, ByzzBench allows access to read the state of the protocol, allowing for the small-scope mutations "in-time" and as well as in value (for example sending a previous prepare message). The rest of the logic for a Zyzzyva replica was implemented by us.

3.1 Why Zyzzyva?

Zyzzyva broke ground on a new type of algorithm that speculatively orders requests sent by clients, cutting down on the prepare phase of other BFT protocols such as PBFT, where replicas agree on the order of execution. In the fast track commit (where a client receives $3f + 1$ responses), Zyzzyva further cuts down on the commit phase and instead moves on to the next request, placing the responsibility of commits onto the client. This was shown to have significant improvements over other protocols such as PBFT in terms of latency and cryptographic operations per request [4]. This performance gain also comes with some increased complexity, for example when reconciling the history during view changes.

According to available literature, ByzzFuzz has never been tested on an algorithm that involves speculation, so the results of the running the testing strategy are particularly useful, especially because the speculation involves added reliance on more state in the protocol. We give ByzzFuzz limited access to the internal state of the replica to see if it uncovers any additional liveness or safety violations. We also would like to understand how ByzzFuzz compares to different testing strategies, such as Twins and if it performs better purely random process and network faults.

Our implementation of Zyzzyva draws from the 2007 paper in which it was introduced [4] as well as its accompanying extended technical report which goes into more detail over the view-change and checkpoint sub-protocols. We extend on ByzzBench's pre-existing abstract replica and leader-based-replica classes which already provide functionalities for message transport over a network, to and from the replica, a commit log for when a replica has a request to commit as well as digest functionality. The replicas each have their own internal state that are made read-only to external parties and communicate with each other only through the sending of messages.

3.2 An Overview of the Zyzzyva Protocol

The Agreement Sub-protocol

The agreement sub-protocol of Zyzzyva has two possible tracks, the fast track and the two-phase track. It is responsible for receiving requests, ordering them and sending back responses with proofs that the system has done so. It also provides ways for the system to recover from issues such as missing requests. If the agreement sub-protocol fails due to a faulty primary, we trigger a view change to replace the faulty primary.

The Fast-Track

The fast track of Zyzzyva occurs when a client receives a matching response from every replica in the system. The fast track is the ideal case in which every node in the system acts without faults. It does this in three steps:

1. The client sends a request r_0 to what it believes is the primary.
2. The primary receives r_0, speculatively orders it and sends a pre-prepare message (ORDER-REQ), o_1 to all other replicas in the system. The pre-prepare message includes the current history hash and r_0.
3. A replica receives o_1, checks that it is well-formed and the history is consistent with its own, executes it and sends a reply (SPEC-RESPONSE) to the client. The client considers the request complete once it has received $3f + 1$ responses.

Once the client receives $3f + 1$ replies from the replicas (every node in the network) it considers the request committed and its place in the system's history irrevocably set. It's important to note here that the replicas do not know if the request has been considered committed by the client as we do not get a response from the client, we only receive the next request.

The Two-Phase track

The two-phase track occurs when the client receives $2f + 1$ responses for a request and takes several more steps for the client to ensure that the request is irrevocably set in the system's history:

1. The client sends a request r_0 to what it believes is the primary
2. The primary receives r_0, speculatively orders it and sends a pre-prepare message, o_1 to all other replicas in the system. The pre-prepare message includes the current history hash and r_0.
3. A replica receives o_1, checks that it is well-formed and the history is consistent with its own, executes it and sends a reply (SPEC-RESPONSE) to the client. In this stage, a replica in this stage does not consider o_1 valid or does not receive the pre-prepare message.
4. The client receives between $2f + 1$ and $3f$ responses by the time its timer expires. It then forms a commit-certificate with the speculative response it has received

and adds the replica's signatures as proof that they agreed to the values. It sends out a commit message (COMMIT) consisting of the commit certificate it formed.

5. A replica receives a commit message, checks that it is valid and if it is, sends back a LOCAL-COMMIT message to the client stating that it has committed the values to its history. Once the client has received $2f+1$ or more matching LOCAL-COMMIT messages, it considers the request complete.

The following parts of the protocol assist in recovery of the system:

Fill-Hole
If a replica receives a higher than expected sequence number from the primary, it could mean that it has fallen behind. If this happens, it will not accept any more messages in the current view until it reconciles its history with the rest of the system. Therefore, Zyzzyva has the fill-hole feature which attempts to fill the holes in a replica's history. Since the literature in which Zyzzyva was introduced lacks many details of the implementation of this, we expand slightly on the feature. It is implemented as follows:

1. A replica receives a sequence number that is higher than expected. It sends out a fill-hole message (FILL-HOLE) with the sequence number (j) that it expected and the sequence number that it received (k). It sends the FILL-HOLE to the primary and sets a timer.
2. If the replica receives any more pre-prepare messages from the primary with a sequence number higher than k, it adds them to the local history log but does not execute them yet.
3. Upon receiving a FILL-HOLE from a replica, the primary fetches the pre-prepare messages from its history and sends them back via a FILL-HOLE-REPLY.
4. Upon receiving every FILL-HOLE-REPLY from j to k (inclusive), the replica cancels the timer, executes the received replies as well as the pre-prepare messages in 2.
5. If the timer expires before the replica receives replies from j to k, then it repeats parts 1 to 4 again but sends the FILL-HOLE to all replicas instead.
6. If the timer expires again, the replica sends a message that the replica is unhappy with the current primary (I-HATE-THE-PRIMARY).
7. The replica also checks if there are any holes from k onwards as this is a possibility. If so, it repeats the fill-hole algorithm.

Forward to Primary
The forward to primary feature forwards an unseen client request from a non-primary replica to the primary. This occurs either when the client doesn't know that the system has gone through a view change and elected a new primary, or when the client resends its request to all replicas after it receives less than $2f$ responses for a given request. It is implemented as follows:

1. The client sends r_0 to a non-primary replica.
2. If the replica already has the request in its cache, it sends the cached speculative response back to the client.
3. If the replica doesn't have the request in its cache, it sends a CONFIRM-REQ message to the primary with the message in order for the primary to order it. It sets a timer and if it doesn't receive back a corresponding ORDER-REQ for the request, it sends an I-HATE-THE-PRIMARY.
4. If the replica receives the ORDER-REQ, it cancels its timer and executes it like normal.

The View Change Protocol
The view change sub-protocol is the sub-protocol that allows the system to elect a new primary once it has deemed the old one faulty. It is as follows:

1. Once a replica has received $f+1$ accusations that the primary is faulty (I-HATE-THE-PRIMARY's) it commits to a view change by sending a message that it has committed to a view change (VIEW-CHANGE) with its current history.
2. Upon receiving $2f+1$ VIEW-CHANGE messages, the future primary then constructs a NEW-VIEW message consisting of a stable version of the system's history. We defer the algorithm for the construction of the history to the technical report accompanying Kotla et al.'s paper.
3. After receiving a NEW-VIEW message, a replica sets its history to the one calculated in the NEW-VIEW message and sends out a VIEW-CONFIRM message.
4. After receiving $2f+1$ VIEW-CONFIRM messages, a replica begins the new view.

It is important to note that after a replica commits to a view change, it does not receive most messages until the new view has begun [4]. We only allow CHECKPOINT messages as well as messages related to the view change to be received while the replica is undergoing the view-change process. We also allow VIEW-CHANGE messages to be received but not processed and instead we send the cached NEW-VIEW message as this would not alter the state of the history in the view change but does allow replicas who missed the first NEW-VIEW message to receive one.

Message	Mutation
$\langle\langle \text{ORDER-REQ}, v, n, h_n, d\rangle_{\sigma p}, m\rangle$	$\langle\langle \text{ORDER-REQ}, \boldsymbol{v'}, n, h_n, d\rangle_{\sigma p}, m\rangle$ $\langle\langle \text{ORDER-REQ}, v, \boldsymbol{n'}, h_n, d\rangle_{\sigma p}, m\rangle$ $\langle\langle \text{ORDER-REQ}, v, n, h_{\boldsymbol{n'}}, d\rangle_{\sigma p}, m\rangle$
$\langle \text{NEW-VIEW}, v+1, P\rangle_{\sigma p}$	$\langle \text{NEW-VIEW}, \boldsymbol{v'}, P\rangle_{\sigma p}$
$\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle_{\sigma i}, i, r, \text{OR}\rangle$	$\langle\langle \text{SPEC-RESPONSE}, \boldsymbol{v'}, n, h_n, H(r), c, t\rangle_{\sigma i}, i, r, \text{OR}\rangle$ $\langle\langle \text{SPEC-RESPONSE}, v, \boldsymbol{n'}, h_n, H(r), c, t\rangle_{\sigma i}, i, r, \text{OR}\rangle$ $\langle\langle \text{SPEC-RESPONSE}, v, n, h_{\boldsymbol{n'}}, H(r), c, t\rangle_{\sigma i}, i, r, \text{OR}\rangle$ $\langle\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t\rangle_{\sigma i}, i, \boldsymbol{r'}, \text{OR}\rangle$ $\langle\langle \text{SPEC-RESPONSE}, v, n, \boldsymbol{h(d)'}, H(r), c, t\rangle_{\sigma i}, i, r, \text{OR}\rangle$
$\langle \text{VIEW-CONFIRM}, v+1, n, h_n, i\rangle_{\sigma i}$	$\langle \text{VIEW-CONFIRM}, \boldsymbol{v'}, n, h_n, i\rangle_{\sigma i}$ $\langle \text{VIEW-CONFIRM}, v, \boldsymbol{n'}, h_n, i\rangle_{\sigma i}$
$\langle \text{VIEW-CHANGE}, v+1, s, [\langle \text{CHECKPOINT}\rangle], CC, O, i\rangle_{\sigma i}$	$\langle \text{VIEW-CHANGE}, \boldsymbol{v'}, s, [\langle \text{CHECKPOINT}\rangle], CC, O, i\rangle_{\sigma i}$ $\langle \text{VIEW-CHANGE}, v, \boldsymbol{s'}, [\langle \text{CHECKPOINT}\rangle], CC, O, i\rangle_{\sigma i}$ $\langle \text{VIEW-CHANGE}, v, s, [\langle \text{CHECKPOINT}\rangle], \boldsymbol{CC'}, O, i\rangle_{\sigma i}$ $\langle \text{VIEW-CHANGE}, v, s, [\langle \text{CHECKPOINT}\rangle], CC, \boldsymbol{O'}, i\rangle_{\sigma i}$

Table 1: Structure aware mutations that we introduce for Zyzzyva. The mutated values are in bold and primed.

Algorithm 1 The Checkpoint sub-protocol

```
1: if $currSeqNum\%CP\_INTERVAL == 0$ then $broadcast(speculativeResponse)$
2: end if
3: if $numSpecResponses \geq 2f+1$ then
4:     $checkpointCC \leftarrow createCommitCertificate()$
5:     $handleCommitCertificate(checkpointCC)$
6: end if
7: $broadcast(checkpointMessage)$
8: if $numCheckpointMessages() \geq f+1$ then
9:     $setCheckpoint(currSeqNum)$
10:    $truncateState()$
11: end if
```

The Checkpoint Protocol

The checkpoint protocol is relatively simple, a replica simply sends out a speculative response every $CP\_INTERVAL$ requests. Upon receiving $2f+1$ matching SPEC-RESPONSE messages, it commits it to its log and sends out a CHECKPOINT message with the current sequence number and history. Upon receiving $f+1$ matching checkpoint messages, it considers the checkpoint stable.

3.3 Message Mutators

In order for ByzzFuzz and our baseline testing strategy to test Zyzzyva, we must also implement mutations for the message which then get applied upon messages that the replica sends. All of our mutations except for the modified history in the VIEW-CHANGE message have small-scope and any-scope variants. We also include increments and decrements for each change in value. For example, the small scope variations for a view number v are $v+1$, $v-1$ and the any-scope mutations are $v+r$ and $v-r$ where r is a random long between 1 and $Long.MAX\_VALUE$ in Java. When mutating values of strings, we generate a random string, such as in the reply of the SPEC-RESPONSE message.

Since the purpose is to test liveness and safety violations in Zyzzyva and not an implementation complete with cryptographic hashes, the hash function we used for the history is XOR. Therefore the way we compute our history is $h_n \leftarrow h_{n-1} \oplus d$ where d is the digest of the message. This has the advantage of being able to compute a previous history from a given history and a message digest, so we can mutate our messages to get the previous history, using the logic $h_{n-1} = h_n \oplus d$ where d is the current digest of the message as this reverses the XOR operation.

We also have read-only access to the replica's state meaning that we can get previous commit-certificates as well as random previous histories, however these are bounded by the history in the log as they get truncated during checkpoints. Finally, we add one extra mutation that is useful in the scenario presented by Abraham et al., the ability to swap the first and last ordered request in the history log during a VIEW-CHANGE message. We consider this to be a small-scope mutation but use it in the testing of any-scope mutations.

A limitation that we encounter here is that ByzzFuzz cannot undo a cryptographic hash function. If we use a cryptographic hash function to calculate the history hash in the ORD-REQUEST and start from an arbitrary previous history (effectively a nonce), it would be nearly impossible to allow the mutators to change the position of the ORDER-REQ's in the local log in the VIEW-CHANGE messages due to the histories not matching up, provided that the 3 criteria of the implementation of ByzzFuzz are met, namely "The implementation of ByzzFuzz requires (i) intercepting the protocol messages exchanged between the processes of the system, (ii) implementing the fault injection algorithm to run on the intercepted messages, and (iii) implementing a set of possible mutations on the protocol messages." according to Winter et al. [9]. If just these criteria are met, ByzzFuzz only has access to h_n in a message and cannot change the values of the history without a replica deeming it invalid. However, in the version of ByzzFuzz implemented on ByzzBench, we have read-only access to the state of the replica and therefore are able to calculate the hash.

4 Experimental Setup and Results

ByzzBench executes the schedules with a given number of network and process faults and records the number of successful scenarios, unsuccessful scenarios and scenarios with errors. We add the test configurations for ByzzBench in the appendix. We run tests with differing levels of process and network faults for a minimum of 400 events (messages sent and timeouts triggered in ByzzBench) as well as a minimum of 100 rounds (each sub-phase in the execution of a request is considered a round). We will give the assignment of the round numbers for each message in the appendix. For reference, while recreating the Abraham safety attack on ByzzBench with ByzzFuzz, it took 86 events, so we expect our tests to be able to capture the attack.

4.1 Baseline

We ran our baseline testing method for 1000 runs on each of the parameters below. We first run it in the case where there are no process faults, and therefore no mutations in the messages in order to establish what number of errors the network faults contribute to. As you can see our implementation of Zyzzyva isn't perfect and has errors in it due to the replica trying to access a position that doesn't exist in the history after failing a validation after a view change. It finds this one error in the 6000 scenarios that were tested.

Due to the limited number of errors and the rarity, we cannot draw concrete conclusions based on which type of scope is better as neither of them found violations in the protocol.

We also created a flawed version of Zyzzyva in which we allow the replica to accept all messages during the view-change process. This means that messages received can interfere with its internal state and lead to safety violations [4]. We do this by not setting the view-change flag (participating in the code) to false when a replica has committed to the view change. We find that the baseline manages to find occasional safety violations. It's interesting to note that the highest number of safety violations were found with no process or network faults. This is presumably due to mutated messages not being considered valid by the system and rejected. Furthermore, there is the case where all replicas receive a correct pre-prepare message from the previous primary, as we haven't changed the view number yet, and a client receives $3f+1$ responses for that request. However, since that request is not in the history logs sent in the VIEW-CHANGE message, it's rolled back and another request takes its position, leading to a safety violation. However, ByzzBench doesn't have the functionality to check application state for a safety violation so we can't test for that.

We receive several errors due to incorrect states of the replica due to the protocol not expecting messages during the view-change and thus creating errors when trying to access something that doesn't exist for example.

Scope	Faults	Safety	Liveness	Error
-	p = 0, n = 0	0	0	0
-	p = 0, n = 7	0	0	1
Small	p = 7, n = 0	0	0	0
Small	p = 15, n = 15	0	0	0
Small	p = 30, n = 30	0	0	0
Any	p = 7 n = 0	0	0	0
Any	p = 15, n = 15	0	0	0
Any	p = 30, n = 30	0	0	0

Table 2: The test results for the baseline testing strategy on our working version of Zyzzyva where p is the number of process faults and n is the number of network faults

Scope	Faults	Safety	Liveness	Error
-	p = 0, n = 0	4	0	92
-	p = 0, n = 7	2	0	8
Small	p = 7, n = 0	2	0	82
Small	p = 15, n = 15	0	0	4
Small	p = 30, n = 30	0	0	5
Any	p = 7 n = 0	0	0	77
Any	p = 15, n = 15	4	0	7
Any	p = 30, n = 30	1	0	6

Table 3: The test results for the baseline testing strategy on the faulty implementation of Zyzzyva which potentially violates safety where p is the number of process faults and n is the number of network faults

4.2 ByzzFuzz

We ran ByzzFuzz on ByzzBench for 1000 runs on each of the parameters below. We were unable to find if we recreated the safety violation presented in Abraham et al. using automated testing due to the way that ByzzBench checks for safety issues. At the end of the safety violation, the commit logs for the replicas have a at position 1, but b has been committed to the application state by c_2, leading to a safety violation but ByzzBench checks only if the replica's commit logs differ, so ByzzBench does not consider it a safety violation, especially given the history of the system is uniform after the second view change that the system goes through.

However, we find that it is possible to recreate it and check manually, the ByzzBench scenario can be found in the code published with this paper. The safety attack requires several precise mutations applied in the right order to messages sent so we further do not expect to find the violation with a small number of scenarios. Here we give a the order of mutations that we applied corresponding to the steps in the safety violation in the paper by Abraham et al [1].

- View 1.2 - Create partition $[i_1, i_2, i_3]$ $[i_4]$
- View 1.3 - Create partition $[i_1, i_2]$ $[i_3, i_4]$, decrement proposed sequence number for ORDER-REQ$_b$. Change history for ORDER-REQ$_b$. Decrement sequence number for SPEC-RESPONSE$_{i_1,b}$. Set first history for SPEC-RESPONSE$_{i_1,b}$.
- View 2.2 - Create partition $[i_1, i_2, i_4]$ $[i_3]$, swap ordered requests in history for VIEW-CHANGE$_{i_1}$, set previous commit certificate for VIEW-CHANGE$_{i_1}$

We consider a case such as this rare to occur in the ByzzFuzz scheduler on ByzzBench and therefore would recommend testing with more scenarios and using an agreement predicate that relies also on the application state.

Scope	Faults	Safety	Liveness	Error
-	p = 0, n = 0	0	0	0
-	p = 0, n = 7	0	0	0
Small	p = 7, n = 0	0	0	0
Small	p = 15, n = 15	0	0	0
Small	p = 30, n = 30	0	0	0
Any	p = 7, n = 0	0	0	2
Any	p = 15, n = 15	0	0	0
Any	p = 30, n = 30	0	0	0

Table 4: The test results for the ByzzFuzz testing strategy on our working implementation of Zyzzyva where p is the number of process faults and n is the number of network faults

ByzzFuzz does not manage to catch any new liveness or safety violations in Zyzzyva, however this could be due to ByzzBench's safety predicate not being able to handle the application state, however in Zyzzyva, safety is partially observed by the client so we do not find safety violations that occur there.

Scope	Faults	Safety	Liveness	Error
-	p = 0, n = 0	6	0	112
-	p = 0, n = 7	4	0	34
Small	p = 7, n = 0	2	0	103
Small	p = 15, n = 15	1	0	22
Small	p = 30, n = 30	0	0	14
Any	p = 7 n = 0	2	0	101
Any	p = 15, n = 15	0	0	128
Any	p = 30, n = 30	0	0	133

Table 5: The test results for the ByzzFuzz testing strategy on the faulty implementation of Zyzzyva where p is the number of process faults and n is the number of network faults

We find that the results for ByzzFuzz for the faulty case look similar to that of the baseline. Comparing the p=0, n=7 entry, we see that round-aware faults are potentially better than purely random network faults. We again find that potentially adding mutations can cause the message to be considered invalid and therefore rejected by the system. Across the values that we chose, we find that ByzzFuzz performs better and small-scope mutations are less likely to have errored executions due to the state, perhaps because they're more readily accepted by the system and therefore less likely cause faults.

4.3 Twins

We ran Twins, implemented on ByzzBench for 5000 scenarios and found no errors or violations in our implementation of Zyzzyva. In order for Twins to recreate the attack, it requires a specific set of partitions which do not surface easily while testing on ByzzBench. This is due to the fact that the partitions change every round, which is far more frequent than the partitions created in the Twins paper where they change partitions once or twice per view [2]. Because of the frequency of changing rounds, we mostly end up stuck with the replicas unable to communicate. Furthermore, 5000 scenarios might not be enough for Twins to catch the safety flaw in Zyzzyva that it was able to in the Bano et al. paper, however, we think this is a suitable amount for our research given the limited resources and that ByzzFuzz and the baseline method manage to find the safety violation several times with a similar number of scenarios.

Safety	Liveness	Error
0	0	0

Table 6: The test results for the Twins testing strategy on our working implementation of Zyzzyva

Safety	Liveness	Error
0	0	1022

Table 7: The test results for the Twins testing strategy on our faulty implementation of Zyzzyva

5 Responsible Research

5.1 Reproducibility

One of the aims of this research is to be reproducible. We will do this by making the following resources available for the results to be reproduced:

1. Implementation details – The code used for the Zyzzyva algorithm with sufficient documentation shall be made publicly available for setup and use.
2. Experimental parameters - The configuration and parameters used in ByzzBench for each one of the testing strategies shall be made available for users to reproduce the results in this paper, this includes for example the ways that messages were mutated in ByzzFuzz as well as general parameters such as the number of messages mutated and dropped, termination criteria and the number of actions per scenario.
3. The scenarios on ByzzBench that result in a liveness or safety violation shall be made available for users to understand and reproduce the issues that Zyzzyva has.

The experiments shall be run on specified hardware and OS configurations which have been detailed in the results section. We attempted to use open source dependencies for the code to make the testing as widely accessible as possible.

5.2 Ethical considerations

Our research aims to try to advance the world of BFT algorithms by providing an evaluation of the detection of Liveness and Safety violations in the algorithms. Since BFT algorithms are used to safeguard against Byzantine faults in distributed systems, which are widely in critical applications, such as cryptocurrencies, it is important to account for the ethical considerations when publishing the results of the experiment.

Potential safety and liveness violations could be exploited by malicious actors to compromise distributed systems, therefore, we follow the practice of Coordinated Vulnerability Disclosure (CVD), the disclosure model in which a vulnerability in a system is only published to the public after relevant parties have had sufficient time to fix the vulnerabilities in their systems. By doing this, we ensure that critical systems have time to be fixed before the results from this experiment can be used to attack them.

6 Discussion

Our results show that there is some difference between ByzzFuzz and the baseline testing method with the former being superior, possibly due to its round-aware fault injection, however the rarity of safety violations makes this difficult to compare given the relatively small sample size of the data. We also find that mutations could also potentially decrease the discovery of some types of violations where incorrect messages are rejected by the system, including small-scope mutations.

6.1 Limitations of ByzzFuzz

While ByzzFuzz's three criteria mentioned before are effective in many cases, we highlight a possible scenario where these criteria prove insufficient for uncovering specific violations. We assume a network wrapper that can does the following:

1. Intercept messages transmitted in the network layer of the test suite.
2. Inject the faults from the fault injection algorithm.
3. Implements a set of mutations for the protocol messages

As part of this definition, the network wrapper does not have access to the actual replicas nor their internal state.
In view 1, part 5, of the first scenario in the Abraham violation c_1 sends a commit-request (COMMIT) to i_1 with a commit certificate for request a. There are three possible scenarios for when i_1 processes the commit certificate in ByzzFuzz: before the view change, during the view change and after the view change. We shall consider each case below:

1. i_1 receives the commit certificate before the view change - If i_1 receives the commit certificate before the view change, it commits a to its commit log and sends a view-change message (VIEW-CHANGE) with the commit certificate denoting a as the first request. This means that i_1 cannot send b as the first request in its log since the history for the commit certificate and b do not match and the view-change message is not considered valid. While computing the ordered request history (G), i_2 accepts a as the first request and a is committed, leading to no safety issues. In order for i_1 to process the commit certificate before the view change and still send a view change with b as the first position in its log, we require the swap first-and-last-histories mutation as previously mentioned as well as a mutation that changes the commit certificate to the previous one (in this case the null commit certificate that the replica is initialized with). However, since the message mutators don't have access to the replica state, it would be nearly impossible for it to generate the previous commit certificate as the mutator has to correctly guess the fields for the certificate which includes a hash for the history.
2. i_1 receives the commit certificate during the view change - after i_1 has sent its commitment to the view change (VIEW-CHANGE), i_1 stops receiving requests until after the new view has been established with a quorum of $2f + 1$ replicas agreeing on the last history hash and sequence number. Therefore, we cannot receive the commit certificate during this time as accepting other messages is considered a safety flaw in Zyzzyva.
3. i_1 receives the commit certificate after the view change - after i_1 receives the new-view message (NEW-VIEW) from i_2, the system, including i_1, agrees that r_2 is the first request in the system's history at sequence number 1. Therefore when i_1 receives the commit certificate for sequence number 1 with the history hash of a, it deems the certificate invalid and doesn't commit a, leading to no safety violations. c_1 continuously resends the COMMIT message to the system until it gets a response, but

since the system doesn't consider its commit certificate valid so it doesn't receive a LOCAL-COMMIT message from the system, forming a potential liveness issue depending on the criteria for liveness.

For an explanation about how Twins overcomes the inability to access state, we defer to paper by Bano et al. in which they introduce twins [2].

7 Conclusions and Future Work

The aim of this research was to evaluate and compare different BFT algorithm testing strategies. There has been groundbreaking work over the past few years with the automation of BFT testing strategies but there is relatively little literature when it comes to the evaluation of these strategies as well as a limited number of BFT protocols tested on them. We evaluated ByzzFuzz, Twins and a baseline testing strategy which randomly injects faults into the network and process in order to see how they would perform with our implementation of Zyzzyva.

One of the aims with the research was to find if ByzzFuzz can uncover faults in the liveness and safety guarantees that Zyzzyva provides. We found that ByzzFuzz does not find the safety violation presented by Abraham et al. in their 2017 paper [1], however it does manage to uncover a safety violation that exists in our faulty implementation of the Zyzzyva. Therefore, we find it has potential to uncover bugs, especially given the relatively small number of simulations that it ran for and the number of times that it found the violation. We would require more testing and a modified agreement predicate to capture the violation presented by Abraham et al. due to the precise nature of mutations and network faults.

We found that ByzzFuzz manages to find more flaws in our implementation of Zyzzyva than the baseline method testing method, presumably due to its round-aware mutations which can possibly simulate byzantine behaviours better, although more research is required into this area. Furthermore, our sample size for testing was on the smaller side (around 6000 simulations) so we cannot draw a conclusive result, however, the results do look promising with the amount of tests run so far. We would recommend that more tests be run with more parameters. We find that ByzzFuzz performs better than Twins when finding the safety violation in the faulty case.

Finally, we wanted to assess the difference between the performance of small-scope and any-scope mutations, for which we cannot draw conclusive results, but it seems that small-scope mutations perform slightly better since they have more cases where the mutations are accepted by the system, however more testing is required to truly understand the implications of small-scope mutations because some violations might require fewer process faults in order to be discovered as shown in our results.

A Appendix

A.1 Testing configurations

Parameter	Value
deliverTimeoutWeight	1
deliverMessageWeight	99
deliverClientRequestWeight	99
dropMessageWeight	15
mutateMessageWeight	15

Table 8: The application.yml test parameters for the baseline testing method

A.2 Round calculations

ORD-REQ	SPEC-RESPONSE
$(s-1)\cdot 10+1$	$(s-1)\cdot 10+2$

Table 9: Round calculations for the ORDER-REQUEST and SPEC-RESPONSE. s is the current sequence number.

COMMIT	LOCAL-COMMIT	CHECKPOINT
$(s-1)\cdot 10+3$	v	$(s-1)\cdot 10+6$

Table 10: Round calculations for the COMMIT, LOCAL-COMMIT and CHECKPOINT. s is the current sequence number and v is the current view number.

IHTP	VIEW-CHANGE	NEW-VIEW	VIEW-CONFIRM
v	v	v	v

Table 11: Round calculations for the I-HATE-THE-PRIMARY, VIEW-CHANGE, NEW-VIEW and VIEW-CONFIRM messages. v is the current view number.

References

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting Fast Practical Byzantine Fault Tolerance. December 2017. arXiv:1712.01367 [cs].

[2] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. Twins: White-glove approach for BFT testing. *CoRR*, abs/2004.10617, 2020.

[3] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Yu Jiang, Ting Chen, and Jiaguang Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2517–2532, 2023.

[4] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine Fault Tolerance.

[5] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3).

[6] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages*, 2(POPL):1–24, January 2018.

[7] Rolando Martins, Rajeev Gandhi, Priya Narasimhan, Soila Pertet, António Casimiro, Diego Kreutz, and Paulo Veríssimo. Experiences with Fault-Injection in a Byzantine Fault-Tolerant Protocol. In David Eyers and Karsten Schwan, editors, *Middleware 2013*, pages 41–61, Berlin, Heidelberg, 2013. Springer.

[8] Atsuki Momose and Jason Paul Cruz. Force-locking attack on sync hotstuff. Cryptology ePrint Archive, Paper 2019/1484, 2019.

[9] Levin N. Winter, Florena Buse, Daan De Graaf, Klaus Von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):757–788, April 2023.