Evaluating the correctness and safety of hBFT with ByzzFuzz

Attila Birke[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: Attila Birke
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Kulahcioglu Özkan, João Miguel Louro Neto, Dr. Jérémie Decouchant

Abstract

Byzantine Fault Tolerant (BFT) protocols are designed to achieve consensus even in the presence of Byzantine faults. Although BFT protocols provide strong theoretical guarantees, bugs in the implementation of the protocols can allow for malicious activity. While previous work, like Twins and Tyr, has focused on alternative methods to test BFT protocols, our work builds upon ByzzFuzz, an automated testing algorithm, which has previously identified bugs in protocols like Tendermint and Ripple. Despite its success, its effectiveness has not yet been tested on speculative BFT protocols like hBFT. This research evaluates the effectiveness of ByzzFuzz in assessing the correctness and safety of hBFT. To address this, we implemented hBFT in ByzzBench, a comprehensive framework where BFT protocols can be evaluated using ByzzFuzz and other testing algorithms. Through testing, ByzzFuzz successfully uncovered a potential violation in hBFT and detected an injected bug in the hBFT implementation. However, detecting the known violation of hBFT required a controlled environment, highlighting areas where ByzzFuzz could be improved. The findings show that ByzzFuzz is effective at identifying bugs in hBFT, while also suggesting the need for improvement to enhance its robustness and adaptability.

1 Introduction

Byzantine Fault Tolerance provides resilience against some Byzantine (malicious) nodes, where a system is Byzantine Fault Tolerant if it has an answer to the Byzantine Generals Problem [1]. BFT aims to ensure reliability and security in distributed systems such as blockchains or cloud computing. The correctness of these systems relies on BFT because if there are bugs in the BFT protocol that the system uses, a malicious party could exploit it, and cause system failures, misbehaviour or even put themselves in an advantageous situation (e.g. fraudulently transferring cryptocurrencies to themselves).

Testing is crucial to ensure the correctness and security of BFT algorithms which previously have been done mostly manually. The bugs, in other words, violations, are usually found years after the protocols are published and put into production. This is due to the lack of sufficient testing methods. For example, a liveness violation [2] was found in 2021 in the read-only optimisation of PBFT [3], more than 20 years after its introduction.

Recent advancements in automated testing for BFT protocols have introduced algorithms like Twins [4], Tyr [5] and ByzzFuzz [6], all of which employ different strategies for identifying potential violations. Twins uses multiple malicious copies of itself, which to other replicas appear identical. On the other hand, Tyr uses a so-called Behaviour Divergent Model for constantly generating consensus messages and making the nodes behave as differently as possible. Both algorithms demonstrated great results, with Twins being used in the production environment of DiemBFT[7] and Tyr finding 20 previously unknown bugs in its tested protocols.

Twins and Tyr provide a great advancement in automated testing for BFT protocols, however, they have only been applied to a limited number of protocols. Our focus is on ByzzFuzz, a randomised testing algorithm that automatically finds errors in the implementation of BFT protocols. ByzzFuzz found multiple bugs in the implementation of PBFT, a potential liveness violation in Tendermint [8] and a previously unknown bug in Ripple [9]. The success of ByzzFuzz comes from the round-based small-scope mutation methods which find bugs that could not be found with other previously used techniques. Although ByzzFuzz was successful at finding bugs in PBFT, Tendermint and Ripple, its effectiveness on protocols beyond remains unexplored.

Another issue in the testing of BFT protocols is the lack of benchmarking tools. As we can see testing algorithms are developed for different BFT protocols - Twins for DiemBFT, ByzzFuzz for Ripple, Tendermint and PBFT - so it is difficult to compare their bug-finding abilities. Although this paper's main goal is to evaluate the effectiveness of ByzzFuzz in finding bugs in the hBFT [10] protocol, it will also contribute to developing a benchmarking tool. This tool is called ByzzBench, and it already has an implementation of PBFT and Ripple as well as the ByzzFuzz and baseline testing algorithms. ByzzBench can be extended to adapt more testing algorithms, e.g. Tyr, and additional BFT protocols. The goal of ByzzBench is to serve as a benchmark where different testing algorithms can be evaluated on the same BFT protocols, and where protocol designers can implement their algorithms and test them with these state-of-the-art testing methods.

hBFT is a speculative protocol, which makes testing it particularly important due to its unique behaviour. Unlike in traditional protocols, in hBFT, replicas may diverge while executing requests and send different replies, to speed up execution and avoid an expensive agreement protocol which would establish order. ByzzFuzz was not tested on speculative protocols previously, so it is even more crucial to test hBFT.

ByzzFuzz provides a state-of-the-art method for testing BFT protocols. To further develop ByzzFuzz and evaluate its performance, additional testing needs to be carried out on more BFT algorithms. Hence, the following paper will focus on testing ByzzFuzz on the hBFT protocol and answer the following question: **RQ: To what extent is ByzzFuzz able to evaluate the correctness and safety of hBFT?**. To help answer the main question, the paper will also answer the following sub-questions:

- **RQ1**: Can ByzzFuzz find any bugs in the implementation of the hBFT protocol?
- **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 hBFT?

2 Related work

The testing of distributed systems has been studied extensively. Previous state-of-the-art work, Jepsen [11], can automatically test distributed systems, and provide great results with its randomised testing method. Despite the great results, Jepsen is a non-Byzantine testing algorithm. Other works, like Netrix [12] and Zermia [13], implement frameworks for testing Byzantine behaviour. Although they are not automatic testing algorithms, they show the importance of testing protocols under unsupervised circumstances.

Automatic testing of BFT protocols is still in its early stages, however, recent works have demonstrated some promising results. Some of the related contributions to our work are Twins and Tyr.

Twins [4], an automated unit test generator, which creates multiple, emulating malicious instances of a process with the same identity, - "twins" - with the same credentials as the normal node. The Twins testing algorithm is designed to test for conflicts in DiemBFT [7]. Using systematic fault injection, Twins can detect three types of faulty behaviours: equivocation, double voting, and losing the internal state. In practice, however, there is a broader range of faults, such as injecting the sequence number of a request with a future value that ByzzFuzz can evaluate.

Tyr [5] uses a behaviour-divergent model that constantly analyses how replicas behave when receiving messages compared to each other. Suppose a replica behaves differently to a specific message. In that case, the algorithm keeps that message in a "message pool", from which the algorithm chooses messages, and will more likely choose this message again. Tyr mutates the messages structurally, like ByzzFuzz, however, it only uses any-scope (arbitrary) message mutations. Tyr uses four oracle detectors to check the safety, liveness, integrity and fairness of the protocols. The latter is unique for Tyr as there are no currently known automatic testing algorithms that test for Fairness consensus property. Tyr covers 297.1% more branches than Twins and found 20 previously unknown bugs in its tested protocols.

Another recent work, BFTDiagnosis [14], focuses on automated performance testing. BFTDiagnosis is an automatic testing algorithm for BFT protocols, however, in contrast to the previously mentioned algorithms, BFTDiagnosis tests the performance of the protocols under Byzantine behaviour. They identified that most of the evaluation of BFT protocols focuses on how fast the protocols are in ideal scenarios, and thus they provide an evaluation under non-ideal circumstances. BFTDiagnosis executes malicious behaviour by implementing different modes - honest and dishonest mode, similar to Tyr - and different Byzantine behaviours (e.g. duplicate, delay). The message mutations are done structurally by mutating the field of the messages. BFTDiagnosis can be an important tool to further evaluate BFT protocols, and see how they perform under malicious behaviour. BFTDiagnosis is similar to ByzzBench as both of these tools give a framework for evaluating either the correctness or the performance of BFT protocols under non-ideal (Byzantine) circumstances.

3 Testing hBFT in ByzzBench

Before conducting any testing on hBFT, the protocol itself needs the be implemented within the ByzzBench framework. There are no current production implementations of hBFT, so the implementation will be based on the hBFT paper [10]. For ByzzFuzz to be able to mutate the messages, the possible mutations will also need to be implemented for hBFT. The mutations will be explained in Section 4.2.

3.1 ByzzFuzz

ByzzFuzz uses round-based, structure-aware, small-scope mutations. *Round-based* means that networks and process faults will be applied in a round manner where even in case of retransmission of a message, the same fault will be applied to the same message. In normal cases, the fault would only be applied to single messages which would mean that the retransmission would be able to get around the fault by delivering the correct message the second time. Although this can be efficient, hBFT does not use many retransmissions. The only case is when the client does not receive enough replies and resends the requests.

Structure-aware refers to the mutation of messages, where the structure of the message will stay the same, but the fields of the message will be mutated. These mutations can be either small-scope or any-scope. We explain the possible mutations for hBFT in Section 4.2.

Small-scope mutations are structure-aware mutations, where the fields of the messages are only changed by a small value. We argue that a small change in the message is more likely to be in the range of acceptable values than changing them arbitrarily. For example, the view number is only incremented or decremented by one, or a single request is removed or added to the speculative history. In the case of digest mutations, the object which the digest is created from is mutated instead of changing the digest by a random value (although in the case of proper digest/hash, a single change in the original message should give a completely different value, similarly to a random change).

Our implementation of ByzzFuzz checks *Integrity* and *Agreement* based on the correctness properties of BFT consensus protocols given in Cachin et al. [15]. *Agreement* checks whether two different replicas have committed different requests at the same sequence number, while *Integrity* checks whether a single replica has committed the same request at different sequence numbers. Our implementation of ByzzFuzz can also detect liveness violations, which checks whether there are any pending messages (including timeouts) that can be processed. In BFT systems liveness means that clients eventually receive replies to their requests, so if there are no more pending messages, the replies will never arrive.

3.2 hBFT

hBFT is a leader-based protocol that uses speculation. This means the replicas can be temporarily out of order, but a subprotocol solves these inconsistencies. In hBFT, it is the three-phase checkpoint sub-protocol. hBFT achieves high performance by having only two stages in the agreement protocol which works the same way for both fault-free and normal

cases. hBFT only requires 3f + 1 replicas for execution, however, this violates the lower bound on the number of replicas needed for a two-stage agreement protocol, set by FaB Paxos [16][17]. Due to this, hBFT breaks safety and has a known violation, more of this in Section 5.

3.3 Methodology

RQ1 will be answered by testing the implemented hBFT protocol through ByzzFuzz using different test parameters. hBFT is known to have a safety vulnerability [17], which should be detected by ByzzFuzz if the algorithm is implemented correctly. If other bugs are found then we need to analyse whether the protocol actually has those violations or if it is just the implementation that is faulty. In either case, it would prove that ByzzFuzz can find bugs, as a defective implementation will most likely have some violations. However, there is the possibility that the protocol has no other violations, and we should not be able to find any other bugs. In this case, bugs will be injected into the protocol intentionally to evaluate the performance of the different testing algorithms.

RQ2 will be answered by running multiple scenarios in ByzzBench with the implemented baseline testing algorithm. After this, the results will be compared to ByzzFuzz. The implementation of the baseline algorithm will allow for the injection of network or process faults arbitrarily. It does not have a restriction to round-based small-scope mutations and we will simulate a Byzantine fault by random any-scope mutations.

RQ3 can be answered by testing the implementation of hBFT by using small-scope mutations (mutating the content of the messages by a small difference) and any-scope mutations (mutating the messages with an arbitrary value). Then the results will be collected, and the difference in the number of violations between the two mutations will be analysed.

4 hBFT in ByzzBench

4.1 hBFT implementation

As there are no existing implementations of hBFT, the current implementation in ByzzBench is purely based on our understanding of the paper. Although the paper explains the execution of the algorithm quite well, there are some parts where the paper is vague and we needed to speculate to be able to code certain parts. As hBFT was introduced after Zyzzyva [18], another speculative protocol, we implemented certain parts based on the Zyzzyva paper where hBFT was too vague (e.g. steps in NEW-VIEW).

The following part is an overview of the implementation and a shorter explanation of where the implementation might differ from the paper due to speculation. For a deeper explanation refer to the paper on hBFT [10]. The notations are described in Figure 2.

The protocol

The protocol consists of 4 major components: agreement, checkpoint, view change, and client suspicion. The agreement takes 2 stages, which can be seen in Figure 1. We assume a correct client at all times, so client suspicion will be skipped. Some of the deviations from the paper are in the commit message, the checkpoint and the view change sub-protocol.

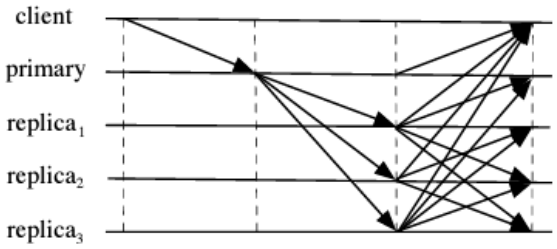

Figure 1: hBFT two-stage agreement protocol

Agreement

Label	Meaning
v	view-number
n	sequence-number
o	operation
t	timestamp
c	client (client ID)
D(x)	digest of x
m	message
δ	speculative execution history
M	also speculative execution history
P	execution history M from CER1(M,v)
Q	execution history from the accepted Checkpoint-I message
R	speculatively executed requests
V	view-change proofs
X	selected checkpoint

Figure 2: Labels given to fields in messages.

In hBFT the agreement sub-protocol consists of two phases, where the primary is responsible for ordering the request. The primary given a REQUEST, <REQUEST,o,t,c>, from the client, sends a PREPARE message, <PREPARE,v,n,D(m),m,c>, to the other replicas. In our implementation, the primary also sends a COMMIT message, <COMMIT,v,n,δ,m,D(m),c>, an alternative would be to consider the PREPARE message from the primary as a COMMIT message. Additionally, the primary also sends a REPLY <REPLY,v,t,n,δ,c>, to the client. If the replicas accept the PREPARE message, they also send a REPLY to the client, a COMMIT message to other replicas and they put the request in the speculatively executed requests[1]. The replicas also send a COMMIT and REPLY if they receive f + 1 matching COMMIT messages. Upon receiving 2f + 1 (including own) COMMIT messages the replica adds the request to the speculative history (commit log) and considers the request committed. If the replica receives f + 1

[1]The speculatively executed requests contain every request that had been executed, even if it is not committed yet.

COMMITs that differ from the PREPARE that it accepted, the replica sends a view change message.

Deviation from the hBFT paper: A small change in our implementation is regarding the acceptance of COMMIT messages. The paper states that COMMIT messages are accepted if the sequence number is either one higher or equal to the replica's sequence number, depending on whether the replica accepted a PREPARE for that sequence number or not. However, this can create a scenario in which requests will not be committed, or the execution will slow down tremendously. Given two clients, all replicas correct, and a request from each client, the primary would PREPARE both requests with sequence numbers 1 and 2 respectively. The replicas then would accept both PREPARE messages and send COMMIT messages. The COMMITS for the first request will never be accepted as they are less than the current sequence number of the replica. Another way to reach a similar problem is if a replica misses both PREPAREs and only receives the COMMITs with sequence number 2, those will not get accepted as they are too ahead of the replica's sequence number.

Checkpoint

The checkpoint sub-protocol consists of 3 steps, which have been implemented according to the paper. The primary is responsible for triggering the checkpoint sub-protocol by sending a CHECKPOINT-I message, <CHECKPOINT-I,n,D(M),M>. If the sequence number or the speculative execution history does not match, the replica sends a VIEW-CHANGE message. Our implementation might differ from the paper as the paper states - "seq is the sequence number of last executed operation" -, however, we use the greatest sequence number in the speculative history. It doesn't necessarily make sense to use the latest executed operation as the main idea behind speculation is that replicas might be out of order, thus executing requests differently. After the replica receives a correct CHECKPOINT-I it broadcasts a CHECKPOINT-II message, <CHECKPOINT-II,n,D(M),M>, to other replicas. Upon receiving 2f + 1 (including own) matching CHECKPOINT-II messages, the replica creates a CER1(M,v). The replica then sends a CHECKPOINT-III message and after receiving 2f + 1 matching, it creates CER2(M,v), at which point the checkpoint is stabilised and previous messages with smaller sequence numbers are discarded.

Possible deviation from the hBFT paper: During the checkpoint sub-protocol, if a replica is missing some requests in its speculative history, due to missing some messages, it needs to adjust to be able to progress ahead. This is not stated in the checkpoint sub-protocol description, however, later in the paper, at 4.4 (2), the following is stated - "correct replicas learn the result and remain consistent". This suggests that upon receiving 2f + 1 CHECKPOINT-II or CHECKPOINT-III messages, the replica is able to copy the speculative history from the CHECKPOINT messages. This is handled by a function call to adjustHistory() for which the pseudocode can be seen at Algorithm 1.

Algorithm 1 adjustHistory

```
M                          ▷ SpeculativeHistory - SortedMap(n, request)
for all n, req in M do
    if n not in History then
        History.put(n, req)
        SpecReq.put(n, req)    ▷ Also add to speculatively executed requests
```

View Change

A view change is responsible for replacing a faulty primary. A replica will send a VIEW-CHANGE message, <VIEW-CHANGE,v+1,P,Q,R>, in any of the following cases: timeout expires for a request; receives f + 1 PANIC messages and a checkpoint is not started within a given time; does not receive a checkpoint after executing a given number of request (primary should trigger the checkpoint after the same threshold); or receives f + 1 view change messages from other replicas. If a replica sends a view change, it stops receiving any messages, other than new-view and checkpoint messages.

Possible deviation from the hBFT paper: The paper doesn't mention any timeouts regarding the view changes, however, the replicas need to be able to move to a higher view if the next primary is also faulty (e.g. in the case of 7 replicas with 2 Byzantine nodes). Based on this reasoning we use the same technique that PBFT uses. When the replica sends a view change, it starts a timer. If the timer expires, then, the replica sends another view change message with a higher view, <VIEW-CHANGE,v+2,P,Q,R>, and starts a new timer with double the timeout as the previous one (1s, 2s, 4s...).

Possible deviation from the hBFT paper: The new primary, upon receiving 2f + 1 view change messages (including own), will construct a NEW-VIEW message, <NEW-VIEW,v,V,X,M>, and broadcast it to all replicas. The paper mentions that a checkpoint protocol is immediately called with M after moving to a new view, however, this is not enough information for the implementation. First of all the M cannot be used as the speculative history for the checkpoint, as replicas may have differing commit-logs (missing some requests), so it will not be accepted in that case. Second, the replicas that have received incorrect requests need to adjust and adapt to the new view requests, or checkpoints. Thus, in our implementation, we do a similar step as in Zyzzyva when receiving a new view message. In short, the replica checks whether the new view message is correct by doing the same calculation as the primary did to construct the new view. After this, the replica executes the requests that the primary proposed in the NEW-VIEW, which can contain possible null requests in which case nothing happens. The idea is that if a request is included in the new view message, then it has been executed by at least 1 correct replica so it can be committed[2]. Lastly, as the new view needs to be succeeded by a checkpoint, we treat the new view as a CHECKPOINT-I message.

[2]If the new primary is faulty, it cannot make a random request committed as at least f + 1 view changes in the view change proofs, which are signed, need to include that request.

4.2 Structure-aware message mutations for hBFT

Message	Mutations
<PREPARE, v, n, d(m), m, c>	<PREPARE, **v'**, n, d, m, c>
	<PREPARE, v, **n'**, d, m, c>
<COMMIT, v, n, d(M), d(m), m, c>	<COMMIT, **v'**, n, d(M), d(m), m, c>
	<COMMIT, v, **n'**, d(M), d(m), m, c>
<CHECKPOINT, n, d(M)>	<CHECKPOINT, **n'**, d(M)>
	<CHECKPOINT, n, d(**M'**)>
<VIEW-CHANGE, v, P, Q, R>	<VIEW-CHANGE, **v'**, P, Q, R>
	<VIEW-CHANGE, v, **P'**, Q, R>
	<VIEW-CHANGE, v, P, **Q'**, R>
	<VIEW-CHANGE, v, P, Q, **R'**>
<NEW-VIEW, v, V, X, M>	<NEW-VIEW, **v'**, V, X, M>
	<NEW-VIEW, v, **V'**, X, M>
	<NEW-VIEW, v, V, **X'**, M>
	<NEW-VIEW, v, V, X, **M'**>

Figure 3: Structure-aware mutations for hBFT.

An overview of the structure-aware mutations can be seen in Figure 3. The variables are the same as in Figure 2. The mutations are similar where the variables are the same. *v* and *n* can be either incremented or decremented. In the checkpoint messages the mutations for *M* include: removing the last or first request, and incrementing or decrementing the sequence number of the last or first request. In each mutation to the speculative history, the mutation must be reflected in the digest as well so it needs to be recalculated. For the view change messages, mutations can be applied to *v*, *P*, *Q*, *R*. *P*, *Q*, *R* can be mutated the following way: remove the last or first request, increment or decrement the sequence number of the last or first request, replace the last or first request with another request but keep the sequence number. In the new view message the view changes *V* can be mutated the following way: remove a view change message, change a view change message to duplicate. The checkpoint *X* can be mutated similarly as in the view change mutations: increment or decrement the checkpoint's sequence number, remove the first or last request. Lastly, the speculative history of the set of requests proposed, *M*, can be mutated the following way: increment or decrement the sequence number of the last or first request, remove the last or first request, add a null request at the end, change the last of first request to null.

5 Manual analysis of the protocol

Before testing the protocol with automated testing methods, we analysed the protocol manually and found the following:

5.1 Known safety violation

hBFT has a known safety violation [17]. This is achieved with a single byzantine primary which equivocates in the PREPARE phase and in its VIEW-CHANGE message. As a result, replicas commit different requests at the same sequence number, within a single view change. The violation is illustrated in Figure 4. We were able to reproduce this violation manually in ByzzBench.

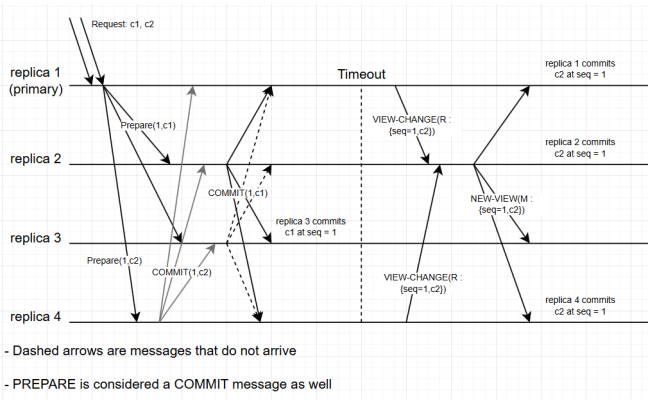

Figure 4: hBFT known violation

Potential violation due to the number of replicas

The number of replicas that hBFT will be tested with is 4. Based on the paper, hBFT should be able to tolerate f Byzantine nodes if the number of replicas is $n \geq 3f + 1$. This would mean 5 replicas could tolerate 1 Byzantine node, however, in Figure 5, we show that hBFT violates safety if $n \neq 3f + 1$. A single primary can equivocate and send a PREPARE with request *m* to f + 1 replicas and with request *m'* to f + 1 other replicas. If we have two partitions: {R1, R2} and {R3, R4} that can only communicate with each other, then it would lead to replicas committing 2 different requests at the same sequence number. This violation, also suggests if n is an arbitrarily large number, a single faulty leader can equivocate by having half of the replicas decide m and the other half decide m'.

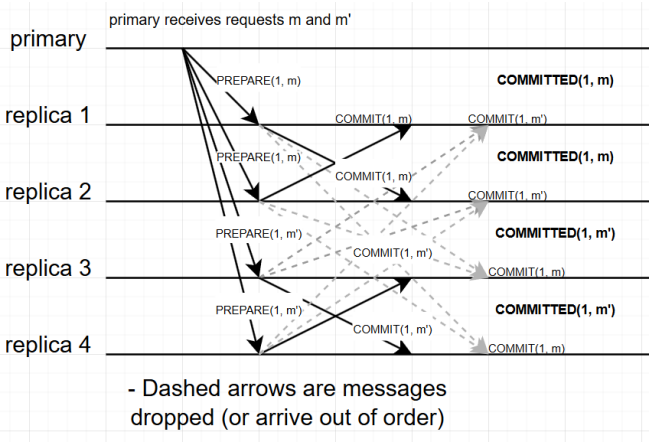

Figure 5: Safety violation with 5 replicas where the primary is Byzantine and equivocates. The COMMIT messages with dashed lines can be dropped (or arrive out of order).

6 Automated testing of hBFT

6.1 Experimental setup

The Checkpoint sub-protocol is important for hBFT as it is responsible for contention resolution. As mentioned before, the Checkpoint protocol is triggered when 2f + 1 PANIC messages are received or when a certain number of requests are executed. This certain number is generally set around 100-150, as a more frequent Checkpoint trigger would slow down the execution. This would also mean that during testing we would need to execute a lot of messages in order to reach this 100 threshold. To speed up the testing process, and to properly evaluate the protocol's correctness through the Checkpoint sub-protocol, the Checkpoint is triggered when the sequence number is divisible by 2 (i.e. is triggered after every 2 requests are completed). This would technically slow down the execution of the protocol, however, we are evaluating the correctness of the protocol and not its speed.

We tested two variants of the implementation. **V1** - the normal execution of the protocol. **V2** - with an implementation bug seeded into the checkpoint message processing. The bug allows checkpoint messages with different digests to be accepted, allowing for a mutated checkpoint to cause a safety violation.

ByzzBench configurations

All different algorithms are run **5000** times, where each run is called a scenario. Each scenario has to execute **500** events before terminating, unless a violation occurs, which terminates the scenario immediately. Events are incremented when a message or a timeout is delivered/executed. Each scenario uses **2** clients and **4** replicas. The tests were executed in **"sync"** mode, which means that instead of randomly picking a message, we chose a random replica and the oldest message in its message log. Additional tests were also executed in **"async"** mode, which randomly picks a message.

Testing

We started our tests with the Random scheduler, responsible for randomly injecting network and process faults into the scenarios. During each scenario, the scheduled messages are executed randomly, and each message can either be delivered, mutated, or dropped based on two parameters. *dropMessageWeight* is responsible for the probability of dropping a message during each step. The *mutateMessageWeight* is responsible for the probability of injecting a random mutation (any-scope) into the message during each step. The higher the weight, the more likely these events will occur. We ran the protocol with the Random scheduler by combining the *dropMessageWeight* and *mutateMessageWeight*. These were set in the range of **[0, 50]**, as seen in Figure 7.

The ByzzFuzz algorithm has 3 parameters. The *numRoundsWithFaults* represents the number of rounds, *N*, over which the faults will be distributed. *numRoundsWithProcessFaults* represents the number of Byzantine faults over *N* rounds. Each process fault simulates a byzantine process fault when delivering messages that contain a round number. If the message matches the sender, receiver and round number of the fault, a random mutation is applied to the message. Similarly, the *numRoundsWithNetworkFaults* represents the number of network faults (message drop) over *N* rounds. Each network fault simulates a network partition when delivering messages that contain a round number. If the round number of the message matches the round number of the fault, and the sender and receiver of the message are not in the same partition, the message will be dropped. These faults are injected when the scenarios are initialised in contrast to the Random scheduler where at each message a decision is made whether it will be delivered, mutated or dropped. As we were more interested in the execution of the protocol under Byzantine behaviour, we ran the ByzzFuzz algorithm with up to 2 number of rounds with network and process faults. The process faults were never set to 0. The faults were injected and distributed over 8 rounds (*numRoundsWithFaults*).

6.2 Results

Potential agreement violation due to hBFT view change

Our tests found a new bug, which leads to an Agreement violation. This was found with all testing methods. The violation comes from the fact that when a replica sends a view change and changes to a disgruntled state, the replica can only accept checkpoint or new view messages. When the replica gets out of the disgruntled state due to correct checkpoint messages, it can continue accepting requests. In this scenario, there is a possibility that the replica commits a request, and a view change happens, where the new primary will not include this committed request in the new view, as it will consider the replicas' old VIEW-CHANGE messages which didn't include this request. This can also happen if the new primary sends the new view, but the replica receives the messages in this order: (1) checkpoint messages to get out of disgruntled state, (2) commit message to commit a request (e.g. R5), (3) new view message. If the other replicas receive the new view before receiving the commit from this replica, they will not commit the request (R5). Overview in Figure 6.

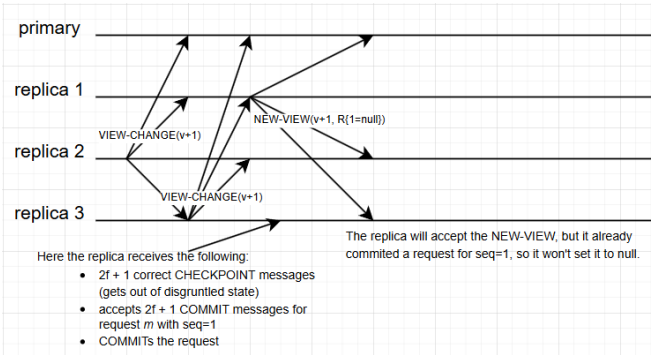

Figure 6: Overview of the violation that is achieved with a checkpoint completed before receiving a new view.

Random scheduler

The random scheduler found one agreement violation at multiple different configurations, which can be seen in Figure 7. The bugs found in version **V1** of the protocol is the violation mentioned in Figure 6. In version **V2** of the protocol, the injected bug was found as seen in Figure 8.

dropMessageWeight	mutateMessageWeight	Agreement violation	Liveness violation
0	0	1	0
25	0	1	0
50	0	0	0
0	25	0	0
0	50	1	0
25	25	0	0
25	50	0	0
50	25	0	0
50	50	0	0

Figure 7: Random scheduler results.

dropMessageWeight	mutateMessageWeight	Agreement violation	Liveness violation
0	25	5	0
0	50	8	0
25	25	2	0
25	50	4	0
50	25	1	0
50	50	1	0

Figure 8: Random scheduler results for protocol version **V2**.

ByzzFuzz algorithm

		Agreement violation		Liveness violation	
		Small scope	Any scope	Small scope	Any scope
N = 0	P = 1	1	0	0	0
N = 0	P = 2	2	1	0	0
N = 1	P = 1	1	0	0	0
N = 1	P = 2	1	0	0	0
N = 2	P = 1	0	0	0	0
N = 2	P = 2	1	1	0	0

Figure 9: ByzzFuzz results with small-scope and any-scope mutations for protocol version **V1**. N is the number of rounds with network faults, and P is the number of rounds with process faults.

Results for V1: As we can see in Figure 9, ByzzFuzz was able to find a number of agreement violations with different input parameters. ByzzFuzz just like the Random scheduler was not able to find any liveness violation, which was expected as hBFT has a lot of different timeouts, which trigger in different scenarios. However, it is important to mention that, the timeout for consecutively increasing view changes was not described in the paper, without which, there might be scenarios where we would get a liveness violation.

The violations, similarly to the Random scheduler, refer to the violation mentioned in Figure 6.

Results for V2: As we can see in Figure 10, ByzzFuzz was able to find a great number of agreement violations with different input parameters. Due to the small-scope mutations, ByzzFuzz was much more effective at exploiting the seeded bug in the implementation, than both the Random scheduler and the any-scope mutations. This was achieved by mutations to the CHECKPOINT messages which change the digest of the message. This made the checkpoint to be accepted even if it did not match other received checkpoints. The requests in the checkpoint were then executed which led to an agreement violation either immediately or a few rounds later.

Some any-scope and some small-scope mutations led to the same previously found violation, mentioned in Figure 6.

		Agreement violation		Liveness violation	
		Small scope	Any scope	Small scope	Any scope
N = 0	P = 1	79	1	0	0
N = 0	P = 2	126	3	0	0
N = 1	P = 1	61	1	0	0
N = 1	P = 2	97	6	0	0
N = 2	P = 1	47	0	0	0
N = 2	P = 2	74	4	0	0

Figure 10: ByzzFuzz results with small-scope and any-scope mutations for protocol version **V2**. N is the number of rounds with network faults, and P is the number of rounds with process faults.

Twins

In addition to ByzzFuzz and baseline methods, we tested hBFT through Twins. We ran the schedule with 1 replica having a "twin", so *numReplicas* was set to 1 and *numTwinsPerReplica* to 2. The results show that Twins were able to find 2 buggy scenarios, and after analysing them, it shows that they are the same violations as previously found with the other methods, Figure 6.

Additional tests

We decided to additionally test the protocol with the checkpoints set to trigger after every 100 requests (or triggered by PANIC or view change). The tests were carried out in the same manner, with the other configurations staying the same but only on version **V1** of the protocol. The Random scheduler and Twins were not able to find any violations in the run scenarios. ByzzFuzz found a few buggy scenarios but the violations were again due to the same checkpoint violation as in previous runs.

		Agreement violation	Agreement violation
		Small scope - "sync"	Small scope - "async"
N = 0	P = 1	0	0
N = 0	P = 2	0	2
N = 1	P = 1	0	0
N = 1	P = 2	3	6

Figure 11: ByzzFuzz results for reproducing the known violation with a forced environment.

We additionally ran tests by forcing the reproduction of the known violation, Figure 11, which meant that only the mutations necessary for the reproduction were implemented. The number of scenarios was set to 20,000 and the number of messages to 50. The violation was found with the expected parameters: in **sync** mode it was found with **N=1** and **P=2**, while in **async** mode with **N=[0,1]** and **P=2**.

7 Responsible Research

Ethical considerations

Considering the ethical and societal impact of research, integrity and reproducibility are essential for any scientific paper. This research focuses on testing hBFT and ByzzFuzz under a simulated environment, and no real-world production implementation has been used. This ensures that no harm is done to any operational systems.

Reproducibility

Regarding the reproducibility of the experiments, the ByzzBench framework was used to implement the testing protocols and the BFT algorithm(s), and conduct any testing. The paper discusses the configurations which the tests were run with. Results are reported truthfully and accurately, however, reproducing the results might deviate from run to run as a random factor needs to be considered when running the algorithms. The configuration and results are clearly stated in the "Experimental setup" and "Results" sub-sections.

The implementation of hBFT stays true to the paper as much as possible, and any deviation from the paper is highlighted and explained. ByzzBench is currently still under development and will become open source including the testing algorithms and BFT protocols implemented in the framework.

Limitations

Regarding the limitations, the paper introduced an implementation of the hBFT protocol, which in certain aspects deviates from its original description. Considering this, the potential violations highlighted in the paper may not affect the original protocol itself. Additionally, in ByzzBench we evaluate the effectiveness of ByzzFuzz based on our implementation, which may also differ from the theoretical design of the algorithm.

Responsible use

In general, conducting a paper where vulnerabilities of a system can be uncovered, might be a concern, however, our results show only a potential violation in our implementation of the hBFT protocol. Our findings do not try to exploit these vulnerabilities but rather improve the reliability of BFT protocols and advance protocol testing. Furthermore, hBFT is not used in a production environment as it breaks safety and is slower than other current protocols used in production. Given further testing on protocols used in production, responsible disclosure should be taken into account when dealing with newly found bugs, but that falls out of the scope of this research.

8 Discussion of Experimental Results

This section provides answers to sub-questions and a thorough reflection of the paper with the aim of answering the primary research question. We answer the questions by summarising and giving a possible explanation of the results of our evaluation of ByzzFuzz and its performance in testing the hBFT protocol:

RQ1: Can ByzzFuzz find any bugs in the implementation of the hBFT protocol? ByzzFuzz managed to uncover a new potential violation caused by inconsistencies during the view changes, as well as the seeded bug in version **V2** of the protocol. Surprisingly, however, ByzzFuzz was not able to find the previously known violation in the normal environment. The reason might be that during normal execution there is a great number of possible mutations and configurations to choose from. In theory, it is possible for ByzzFuzz to find the violation as it only requires 2 mutations in 2 different rounds, thus, given a higher number of runs, there is an almost certain possibility that the bug would be found. This is further supported by the fact that the violation was found by ByzzFuzz in a controlled (forced) environment.

RQ2: How does the bug detection performance of ByzzFuzz compare to a baseline testing method that arbitrarily injects network and process faults? ByzzFuzz has provided similar results as baseline testing methods in the normal **V1** version of the protocol, as both managed to find the new potential violation. This is due to the fact that the new potential violation only requires an unreliable (asynchronous) network, which can be reached by dropping or delivering messages out of order. In the seeded version **V2** of the protocol, ByzzFuzz found the violation with a much higher likelihood due to its small-scope mutations.

RQ3: How do small-scope and any-scope message mutations of ByzzFuzz compare in their performance of bug detection for hBFT? Both small-scope and any-scope mutations were able to recognise the same potential violation in the normal version, **V1**, of the protocol. In version **V2**, the seeded bug was mostly exposed by small-scope mutations, and although it can be detected by any-scope mutations, it has a lower likelihood. That is, provided that any-scope mutations trigger the right message mutation (e.g. decrementing/incrementing a sequence number by a correct amount). Additionally, due to similar aforementioned reasons, in the forced execution, only the small-scope mutations managed to find the known safety violation, which further proves the strength of small-scope mutations in exposing critical implementation bugs.

Regarding previous work, ByzzFuzz was compared to Twins. The results show that ByzzFuzz was more effective at finding both the known violation - in a controlled environment - and the seeded bug in version **V2** of the protocol. The reason is that in contrast to ByzzFuzz, Twins do not use structure-aware mutations which leads to it missing the required steps in exploiting the violations.

We believe ByzzFuzz and ByzzBench can be used for testing in a production environment. ByzzBench is an adequate framework for implementing and testing different testing algorithms on multiple BFT protocols. Based on this paper and previous work on ByzzFuzz, it is clear that it has the ability to find bugs in BFT protocols that normal, baseline testing methods cannot. However, based on the results, further refinement of the algorithm is necessary to provide better coverage of protocols. Potentially, it could be combined with other algorithms, such as Twins or Tyr, but that is for future research.

9 Conclusions and Future Work

Conclusion

This paper presents an evaluation of ByzzFuzz on hBFT, a speculative BFT protocol, alongside our implementation of both ByzzFuzz and hBFT.

ByzzFuzz is effective at discovering fault-tolerance bugs in the implementation of hBFT. It was also successful at discovering a seeded bug in a modified version of hBFT, which was

found less likely with baseline methods and not at all with the Twins algorithm. This shows ByzzFuzz's strengths of its round-based, small-scope, structure-aware mutations.

Compared to baseline testing, ByzzFuzz has proven more effective at finding violations. In the normal version of the protocol, both baseline and ByzzFuzz testing managed to find the new potential violation. When testing with the bug-injected version of hBFT, ByzzFuzz was able to discover the injected bug with a much higher likelihood.

Comparing the small-scope and any-scope mutations showed that small-scope mutations were more effective at finding violations. Given the bug-injected version of hBFT, the small-scope mutations discovered the bug with a much higher probability. Additionally, in a controlled environment, only the small-scope mutations were able to find the known violation.

Moreover, through manual analysis, we identified a potential violation related to hBFT's assumption of the number of replicas required for Byzantine Fault Tolerance. This further extends our understanding of hBFT's limitations and highlights areas for protocol improvement.

This research also contributes towards the aim of creating a benchmarking tool for evaluating automatic testing algorithms by developing ByzzBench.

To conclude, this research highlights the effectiveness of ByzzFuzz as a structured testing tool for uncovering subtle vulnerabilities in BFT protocols. By combining targeted fault injection and precise mutation strategies, ByzzFuzz offers a reliable approach to BFT protocol evaluation.

Future work

So far ByzzFuzz has been tested on two production implementations, namely Tendermint and Ripple, where it found multiple violations. Although ByzzFuzz was successful in finding these bugs, it is important to add further methods for testing, as it had trouble with finding the violation in hBFT.

ByzzFuzz can be further improved within our benchmark suite, ByzzBench. For example, the current implementation of ByzzFuzz in ByzzBench does not check for "bounded liveness" - or "bounded termination" - which considers whether a consensus is reached within a bounded amount of time. With such further improvements, we could improve the testing algorithm, and provide coverage for more violations.

A potential future work includes combining the strengths of Tyr and ByzzFuzz. ByzzFuzz's strength lies in its round-based small-scope mutations, while Tyr profits from its Behaviour Divergent Model which analyses the behaviour of nodes. Although this would not improve ByzzFuzz itself as it would create a new testing algorithm, it would have the potential to achieve better results than the two separately.

References

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 382–401, 1982.

[2] C. Berger, H. P. Reiser, and A. Bessani, "Making reads in bft state machine replication fast, linearizable, and live," 2021.

[3] 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.

[4] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," 2022.

[5] 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, 2023.

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

[7] T. D. Team, "Diembft v4: State machine replication in the diem blockchain," 2021.

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

[9] D. Schwartz, N. Youngs, and A. Britto, "The ripple protocol consensus algorithm," 2014.

[10] S. Duan, S. Peisert, and K. N. Levitt, "hBFT: Speculative Byzantine Fault Tolerance with Minimum Cost," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 1, pp. 58–70, 2015.

[11] K. Kingsbury, "Jepsen," 2022.

[12] C. Dragoi, C. Enea, S. Nagendra, and M. Srivas, "A domain specific language for testing consensus implementations," 2023.

[13] J. Soares, R. Fernandez, M. Silva, T. Freitas, and R. Martins, "Zermia - a fault injector framework for testing byzantine fault tolerant protocols," in *Network and System Security* (M. Yang, C. Chen, and Y. Liu, eds.), (Cham), pp. 38–60, Springer International Publishing, 2021.

[14] J. Wang, B. Zhang, K. Wang, Y. Wang, and W. Han, "Bftdiagnosis: An automated security testing framework with malicious behavior injection for bft protocols," *Computer Networks*, vol. 249, p. 110404, 2024.

[15] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd ed., 2011.

[16] J.-P. Martin and L. Alvisi, "Fast byzantine consensus," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 3, pp. 202–215, 2006.

[17] N. Shrestha and M. Kumar, "Revisiting hbft: Speculative byzantine fault tolerance with minimum cost," *CoRR*, vol. abs/1902.08505, 2019.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," *ACM Trans. Comput. Syst.*, vol. 27, no. 4, 2010.

A Almost-MAC agreement

In the hBFT paper, the Almost-MAC agreement serves as a protection against a faulty primary removing a correct client. In the normal execution of the protocol, a faulty primary can trigger a checkpoint through PANIC messages, which would get sent by the client in case it detects the primary to be faulty (gets more than f but less than 2f + 1 replies). If the primary is faulty and doesn't do anything else, it will be replaced through a view change that takes place during the checkpoint sub-protocol. However, if the primary colludes and makes the request committed in the checkpoint sub-protocol (sends a reply to the client and includes the request in the CHECKPOINT message), the execution will continue and the replicas that called the PANIC will suspect the client to be faulty. A possible solution to this is the Almost-MAC agreement which includes the PREPARE message in the COMMIT messages. This means that if a single correct replica accepts a well-formatted PREPARE message, it can make the request executed. This of course also increases the amount of cryptographic operations in the agreement protocol as the message will become bigger, and will generally slow down the execution.

In our testing environment, we don't consider faulty clients to be present, thus using the Almost-MAC agreement just for the reason of avoiding correct clients being suspected faulty is unreasonable. Nevertheless, we implemented the Almost-MAC agreement as an alternative to the normal agreement protocol. The changes in the Almost-MAC agreement protocol are the following:

- PREPARE messages are appended to COMMIT messages
- If a COMMIT message is accepted by a replica, and the appended PREPARE is correct, the replica also executes the request
- If a replica accepted a PREPARE message and receives a correct COMMIT message but with a conflicting PREPARE[3] message it starts a view-change

B Additional test results

		Agreement violation		Liveness violation	
		Small scope	Any scope	Small scope	Any scope
N = 0	P = 1	0	0	0	0
N = 0	P = 2	0	0	0	0
N = 1	P = 1	0	0	0	0
N = 1	P = 2	1	0	0	0
N = 2	P = 1	1	0	0	0
N = 2	P = 2	1	1	0	0

Figure 12: ByzzFuzz "sync" results with checkpoint triggering set to 100 requests. All violations are the new potential violation found.

C Pseudo codes

[3]PREPARE messages are signed by the primary, so a faulty replica could not change the contents of the appended PREPARE message

		Agreement violation		Liveness violation	
		Small scope	Any scope	Small scope	Any scope
N = 0	P = 1	0	0	0	0
N = 0	P = 2	1	1	0	0
N = 1	P = 1	1	0	0	0
N = 1	P = 2	3	0	0	0
N = 2	P = 1	1	1	0	0
N = 2	P = 2	1	0	0	0

Figure 13: ByzzFuzz "async" results with checkpoint triggering set to 2 requests. All violations are the new potential violation found.

dropMessageWeight	mutateMessageWeight	Agreement violation	Liveness violation
0	0	1	0
25	0	1	0
50	0	0	0
0	25	1	0
0	50	1	0
25	25	0	0
25	50	0	0
50	25	0	0
50	50	0	0

Figure 14: Random "async" results with checkpoint triggering set to 2 requests. All violations are the new potential violation found.

dropMessageWeight	mutateMessageWeight	Agreement violation	Liveness violation
0	0	1	0
25	0	1	0
50	0	1	0
0	25	0	0
0	50	0	0
25	25	0	0
25	50	1	0
50	25	0	0
50	50	1	0

Figure 15: Random "async" results with checkpoint triggering set to 100 requests. All violations are the new potential violation found.

		Agreement violation	Liveness violation
		Small scope	Small scope
N = 0	P = 1	0	0
N = 0	P = 2	0	0
N = 1	P = 1	0	0
N = 1	P = 2	3	0

Figure 16: ByzzFuzz results with forcing the reproduction of the known violation. Each configuration was run 20,000 times in **"sync"** mode, and minNumOfEvents was set for 50 as the violation should happen within only 15 messages. Additionally, only the required mutations for the violation were implemented.

		Agreement violation	Liveness violation
		Small scope	Small scope
N = 0	P = 1	0	0
N = 0	P = 2	2	0
N = 1	P = 1	0	0
N = 1	P = 2	6	0

Figure 17: ByzzFuzz results with forcing the reproduction of the known violation. Each configuration was run 20,000 times in **"async"** mode, and minNumOfEvents was set for 50 as the violation should happen within only 15 messages. Additionally, only the required mutations for the violation were implemented.

Algorithm 2 Simple overview of Random scheduler decision making.

```
timeoutWeight *= #ofTimeouts
deliverMessageWeight *= #ofMessages
deliverClientReqWeight *= #ofReqs
dropMessageWeight *= #ofMessages
mutateMessageWeight *= #ofMessages

weights = ∑ Weights
dieRoll = random(0, weights)

if dieRoll -= timeoutWeight < 0 then
    deliverTimeout()
else if dieRoll -= deliverMessageWeight < 0 then
    deliverMessage()
else if dieRoll -= deliverClientReqWeight < 0 then
    deliverClientReq()
else if dieRoll -= dropMessageWeight < 0 then
    dropMessage()
else if dieRoll -= mutateMessageWeight < 0 then
    mutateMessage()
```