Randomized Testing of Byzantine Fault Tolerant Algorithms

Winter, Levin N.; Buşe, Florena ; de Graaf, Daan ; von Gleissenthall, Klaus; Kulahcioglu Ozkan, Burcu

DOI
10.1145/3586053

Publication date
2023

Document Version
Final published version

Published in
PACMPL

Randomized Testing of Byzantine Fault Tolerant Algorithms

LEVIN N. WINTER, Delft University of Technology, The Netherlands
FLORENA BUSE, Delft University of Technology, The Netherlands
DAAN DE GRAAF, Delft University of Technology, The Netherlands
KLAUS VON GLEISSENTHALL, Vrije Universiteit Amsterdam, The Netherlands
BURCU KULAHCIOGLU OZKAN, Delft University of Technology, The Netherlands

Byzantine fault-tolerant algorithms promise agreement on a correct value, even if a subset of processes can deviate from the algorithm arbitrarily. While these algorithms provide strong guarantees in theory, in practice, protocol bugs and implementation mistakes may still cause them to go wrong. This paper introduces ByzzFuzz, a simple yet effective method for automatically finding errors in implementations of Byzantine fault-tolerant algorithms through randomized testing. ByzzFuzz detects fault-tolerance bugs by injecting randomly generated network and process faults into their executions. To navigate the space of possible process faults, ByzzFuzz introduces *small-scope* message mutations which mutate the contents of the protocol messages by applying small changes to the original message either in value (*e.g.*, by incrementing the round number) or in time (*e.g.*, by repeating a proposal value from a previous message). We find that small-scope mutations, combined with insights from the testing and fuzzing literature, are effective at uncovering protocol logic and implementation bugs in real-world fault-tolerant systems.

We implemented ByzzFuzz and applied it to test the production implementations of two popular blockchain systems, Tendermint and Ripple, and an implementation of the seminal PBFT protocol. ByzzFuzz detected several bugs in the implementation of PBFT, a potential liveness violation in Tendermint, and materialized two theoretically described vulnerabilities in Ripple's XRP Ledger Consensus Algorithm. Moreover, we discovered a previously unknown fault-tolerance bug in the production implementation of Ripple, which is confirmed by the developers and fixed.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computing methodologies → Distributed algorithms**.

Additional Key Words and Phrases: Distributed consensus, Byzantine fault-tolerance, Random testing

ACM Reference Format:
Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 101 (April 2023), 32 pages. https://doi.org/10.1145/3586053

1 INTRODUCTION

Byzantine fault-tolerant consensus algorithms help replicate services on a set of distributed nodes in a fault-tolerant manner, even if a subset of nodes deviates from the protocol arbitrarily. These algorithms are at the core of many distributed systems, including consortium-based blockchain

Authors' addresses: Levin N. Winter, Delft University of Technology, Delft, The Netherlands, l.n.winter@student.tudelft.nl; Florena Buse, Delft University of Technology, Delft, The Netherlands, i.buse@student.tudelft.nl; Daan de Graaf, Delft University of Technology, Delft, The Netherlands, D.J.A.deGraaf@student.tudelft.nl; Klaus von Gleissenthall, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, k.freiherrvongleissenthal@vu.nl; Burcu Kulahcioglu Ozkan, Delft University of Technology, Delft, The Netherlands, b.ozkan@tudelft.nl.

2475-1421/2023/4-ART101
https://doi.org/10.1145/3586053

systems, such as Tendermint [Buchman 2016], Hyperledger [Androulaki et al. 2018], Libra [Baudet et al. 2019], Ripple [Schwartz et al. 2014] and Stellar [Mazieres 2015].

To ensure the correct functioning of these systems, it is crucial that consensus algorithms are designed and implemented correctly. Mistakes can lead to serious reliability and security problems, such as forking a blockchain cluster, in which rogue nodes can commit conflicting transactions allowing them to spend their funds twice or allowing attackers to render the entire system inoperable.

In principle, Byzantine fault-tolerant (BFT) algorithms offer strong protection against such attacks. In practice, however, making sure that they are designed and implemented correctly is difficult. Even seemingly minor mistakes may compromise the algorithm's core guarantees in executions with subtle network and process faults. Therefore, it comes as no surprise that a string of recent theoretical analyses of BFT algorithms discovered several adversary scenarios that violate correctness [Abraham et al. 2017; Amores-Sesar et al. 2020; Amoussou-Guenou et al. 2019; Berger et al. 2021; Kim et al. 2019; Mauri et al. 2020; Momose 2019], suggesting many more unknown cases.

Despite these concerns, we surprisingly still lack automated methods for checking Byzantine fault-tolerant system implementations. Most existing techniques target concurrency, partition fault tolerance, or benign process fault tolerance bugs.

The common practice for testing large-scale distributed systems is randomized testing, which offers a practical way of searching for bugs. Practitioners test their systems by randomly introducing network faults to isolate processes or partition the network [Kingsbury. 2022] or by using a collection of perturbation tools (dubbed monkeys) that randomly inject a set of faults into the executions of production systems [Bennett and Tseitlin 2012; Hadoop. 2009; Pogde et al. 2020; Rosenthal 2017]. These techniques have been shown to uncover many bugs in large-scale distributed systems successfully. Moreover, recent work [Majumdar and Niksic 2018] provided a theoretical explanation for the success of random testing in detecting partition fault tolerance bugs. However, these techniques fail to explore the system's behavior under arbitrary or malicious process behavior and, therefore, cannot detect Byzantine fault tolerance bugs.

In this paper, we aim to address the need for a practical testing tool for checking Byzantine fault-tolerant systems against arbitrary process faults. We present ByzzFuzz, a method for finding fault-tolerance bugs in the implementations of consortium-based Byzantine fault-tolerant consensus algorithms. Motivated by the effectiveness of random testing for detecting network and benign process fault tolerance bugs, ByzzFuzz aims to detect Byzantine fault tolerance bugs in a randomly generated set of test executions with arbitrary process faults.

To navigate the enormous space of possible process faults, ByzzFuzz introduces the notion of *small-scope mutations* to protocol messages which, together with three key insights from the testing and fuzzing literature, i.e., *fault-bounded testing*, *round-based fault injection*, and *structure-aware mutations*, allow us to uncover deep logic and implementation bugs in real-world protocols.

Fault-bounded testing. First, ByzzFuzz samples from the set of system executions only with a bounded number of *network faults*, which may drop messages, and *Byzantine process faults*, which may omit or corrupt messages. Motivated by the effectiveness of bounded testing approaches such as context bounding [Musuvathi and Qadeer 2007], delay bounding [Emmi et al. 2011], and bug depth bounding [Burckhardt et al. 2010; Kulahcioglu Ozkan et al. 2018], we bound the number of faults in an execution. Fault-bounding captures the intuition that a high number of fault-tolerance bugs manifest in the existence of a small number of deviations from the synchrony of the network and the correct functioning of the processes.

Round-based testing. Second, following previous work on the correctness analysis of crash-fault consensus algorithms [Aminof et al. 2018; Bertrand et al. 2019; Damian et al. 2019; Dragoi et al.

2020, 2014; Tsuchiya and Schiper 2011], we structure both network faults and Byzantine process faults in terms of the communication rounds in the protocols. Hence, we do not lose or mutate individual messages, but we drop or mutate all messages sent in a randomly chosen communication round. Round-based fault injection has several advantages: First, it allows us to deal with message retransmissions effectively. If a single message is lost, the sender process will often try resending it, rendering the injected network fault ineffective. In contrast, round-based network errors will drop all messages corresponding to the logical protocol step. Similarly, round-based process faults allows us to send the same incorrect message to multiple processes in the same round, making it more likely to produce split-brain scenarios. Second, round-based fault injection makes it easier to reproduce a faulty execution and analyze it for debugging. Regardless of the delivery order of messages, ByzzFuzz injects the faults in the same execution rounds given the same random seed. Finally, round-based fault injection reduces the sample set of executions and hence increases the probability of hitting a specific execution [Dragoi et al. 2020].

Structure-aware mutations. Third, ByzzFuzz uses structure-aware mutations to corrupt the messages sent by a Byzantine process. Corrupting a message by applying bit-level corruptions on the serialized representation of a message is likely to corrupt the message's formatting, leading to the discovery of deserialization and parsing bugs. However, by only exercising parsing logic, these corruptions fail to test deeper implementation logic behavior that occurs upon receipt of the tampered message. Inspired by the high-level mutations used in input fuzz testing [Padhye et al. 2019; Park et al. 2020; Pham et al. 2021; Zhang et al. 2020], we mutate messages by either omitting them or modifying them using high-level, structure-preserving mutations. High-level mutations provide syntactically valid protocol messages that deviate from normal protocol execution (e.g., a syntactically correct protocol message with an incorrect sequence number), allowing us to discover deep implementation logic bugs.

Small-scope mutations. Finally, we introduce *small-scope mutations*, which mutate the *contents* of the protocol messages by applying small changes to the original values of the message fields. Inspired by the small-scope hypothesis for testing [Jackson and Damon 1996], we argue that a high proportion of fault-tolerance bugs can be found via small modifications of the original messages, either in value (*e.g.*, by incrementing the round number) or in time (*e.g.*, by repeating a proposal value from a previous message). Our intuition for small-scope mutations is twofold. First, incorrect values close to the correct value are more likely to remain in the valid interval of expected values and, therefore, not to be ignored. Second, they capture boundary values between the protocol rounds. For example, increasing/decreasing an index number can correspond to the correct value to a previous/following protocol round, and it is more likely to exercise faulty protocol logic than replacing it with an arbitrary number (see Section 3).

Contributions. Based on the key ideas above, we implemented ByzzFuzz[1] and applied it to test the implementations of two popular blockchain systems, Tendermint and Ripple, as well as the classic PBFT protocol. Our experiments show that ByzzFuzz's combination of a bounded number of structure-aware, small-scope message mutations, and round-based testing yields the first practical tool that can detect previously unknown bugs in production blockchain systems. In our evaluation, all the protocol bugs discovered via arbitrary mutations could also be found via small-scope mutations. Moreover, some of the violations require small-scope mutations and cannot be discovered by adding larger perturbations, including a new bug we detected in Ripple. ByzzFuzz detected a known protocol vulnerability in PBFT, several bugs in an implementation of the protocol we found on GitHub, as well as several executions with a potential violation of termination in Tendermint. ByzzFuzz also materialized two previously known protocol vulnerabilities in

[1]The source code of ByzzFuzz is available in the accompanying artifact [Winter et al. 2023].

Ripple [Amores-Sesar et al. 2020], where ByzzFuzz exposed a new scenario which, unlike the known one, subverts Ripple's detection of Byzantine nodes, allowing the Byzantine process to go undetected. Finally, ByzzFuzz discovered a previously unknown implementation bug in the production implementation of Ripple. To sum up, our contributions are the following:

- We present ByzzFuzz, a randomized test case generator for detecting bugs in Byzantine fault-tolerant algorithms. The novelty of ByzzFuzz lies in our integration of several insights for tackling the sample set of process faults and building an automated testing algorithm that is simple yet effective at detecting fault-tolerance bugs. As far as we are aware, ByzzFuzz is the first automated testing tool that managed to discover previously unknown Byzantine fault-tolerance bugs in production blockchain systems.
- We propose *small-scope* semantic mutations to protocol messages for modeling arbitrary process faults and empirically show that they are effective at discovering the vulnerabilities of Byzantine fault-tolerant consensus systems.
- We materialized two previously known protocol vulnerabilities in Ripple and discovered a previously unknown implementation bug in the production implementation of Ripple, which is confirmed by the developers and fixed.

2 BYZANTINE FAULT TOLERANT ALGORITHMS

In this section, we provide our assumptions for the distributed system model and briefly describe Byzantine fault-tolerant consensus algorithms together with their correctness properties.

2.1 System Model

We consider a system with a set of distributed processes and an arbitrary number of clients that communicate over a network. We assume the Byzantine process fault model, *i.e.*, a faulty process can deviate arbitrarily from the protocol specification [Lamport et al. 1982]. We refer to the Byzantine processes as *faulty* and the other processes as *correct*.

We limit the strength of a faulty process by assuming that it is computationally bounded and unable to break the cryptographic algorithms employed in the protocol. Therefore, an adversary process: *(i)* cannot guess a random number and cannot decrypt messages unless it has the key, *(ii)* cannot produce a valid signature of another process, *(iii)* can only send messages with its own identity, *(iv)* cannot compute the information hashed into a digest, and *(v)* cannot compute two different messages producing the same digest.

Our testing approach assumes that the network satisfies the assumptions of the distributed system under test. In this work, we focus on Byzantine fault tolerant consensus algorithms, in particular, PBFT, Tendermint, and Ripple, which assume a variant of a partially synchronous network [Dwork et al. 1988] that behaves eventually synchronous. Therefore, we assume the processes communicate over an eventually synchronous network can behave asynchronously, *i.e.*, arbitrarily delay or drop the messages for some duration of time. However, it eventually behaves synchronously and delivers all the messages within a fixed time-bound, which is unknown to the processes. Recovery of the network after some periods of asynchrony avoids the famous FLP impossibility result [Fischer et al. 1985] for asynchronous networks, which can indefinitely delay messages. The model provides a realistic network assumption widely accepted in the design of distributed consensus algorithms [Cachin and Vukolic 2017].

2.2 Byzantine Fault Tolerant Consensus

Distributed consensus algorithms (or protocols) define a set of rules that determine how the cluster of processes reaches agreement for choosing a common value out of a set of proposed

values. Byzantine fault tolerant (BFT) protocols achieve agreement in the presence of Byzantine or malicious processes that display arbitrary behavior.

Consensus protocols are typically designed as a sequence of *communication rounds* in each of which the processes exchange some messages regarding some steps of the decision process. In each round,[2] the processes send messages, then receive messages that are sent to them and update their local states accordingly. Typically, a proposer (or leader) node starts the protocol execution by proposing a transaction to commit with a specific sequence number. After the leader broadcasts the proposal, the participants exchange messages to vote for the proposal. As an example, we provide a brief overview of the seminal PBFT algorithm.

The PBFT Algorithm. Practical Byzantine Fault Tolerance (PBFT) [Castro and Liskov 1999] is a seminal consensus algorithm that provides distributed agreement in a cluster of processes in the existence of malicious (or Byzantine) processes. PBFT tolerates f Byzantine faulty processes, which can deviate arbitrarily from the protocol specification, in a network of at least $3f + 1$ processes. An execution of PBFT is decomposed into *view*s in each of which one of the processes acts as the leader. In each view, the leader executes a sequence of client operations. For each client request, the leader broadcasts a proposal, which is followed by two rounds of message exchanges in which the participants vote for the proposal. If they make agreement on the proposal, they process the operation and return the reply to the client. The processes keep the total order of executed client operations in their message logs. PBFT ensures that the correct processes agree on their logs.

Figure 1a shows the normal case execution of PBFT where the cluster processes a client request in the fault-free scenario. When a client requests an operation by sending a `REQUEST`, the leader process (here p0) collects some rounds of votes from the processes to issue the requested operation.

In the `PRE-PREPARE` round, the leader assigns the sequence number n to the request m and multicasts it in a message $\langle \langle \texttt{PRE-PREPARE}, v, n, d \rangle , m \rangle$, containing the current view number v, the sequence number n, and a digest d of the request, which serves as a checksum. A process accepts `PRE-PREPARE` if the message is in the current view v, the sequence number is in the expected interval, and the process has not accepted another request for the same sequence number. The processes that accept `PRE-PREPARE` move to the `PREPARE` round and multicast $\langle \texttt{PREPARE}, v, n, d, i \rangle$, where i is the sender's process-id. A process collecting at least $2f + 1$ matching `PREPARE` messages multicasts $\langle \texttt{COMMIT}, v, n, d, i \rangle$ to all processes. When a process collects $2f + 1$ matching commit messages, it executes the request m and sends `REPLY` to the client. A request is considered to be completed when the client receives at least $f + 1$ matching replies from different replicas. The protocol offers an optimization for improving the performance of read-only operations, which requires the clients to receive at least $2f + 1$ matching responses to accept a result.

If a process suspects that the current leader in view v is faulty, *e.g.*, when it detects a time-out while executing a request, the process can start a `VIEW-CHANGE` round to select the new leader for view $v + 1$. We omit the description of the `VIEW-CHANGE` and `NEW-VIEW` rounds for brevity.

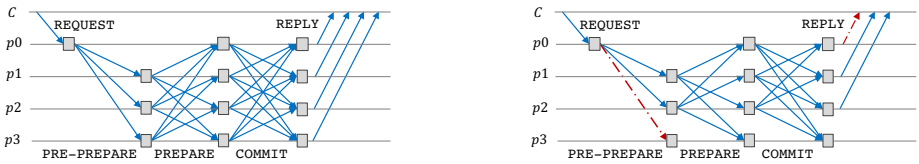
2.3 Properties of Byzantine Consensus

The correctness properties of BFT consensus protocols are given as [Cachin et al. 2011]:

- *Termination.* Every correct process eventually decides on some value.
- *Validity.* A correct process may only decide a value that was proposed by a correct process.
- *Integrity.* No correct process decides twice.
- *Agreement.* No two correct processes decide differently.

The correct design and implementation of BFT consensus algorithms ensure the safety properties of *validity*, *integrity*, and *agreement* in the executions with a certain amount of faulty processes.

[2] Also referred to as a *phase* or *step* in some protocols.



(a) The normal case execution.

(b) A liveness violation.

Fig. 1. The normal case execution and an execution violating liveness of the PBFT protocol.

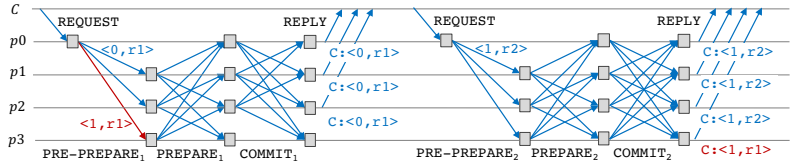


Fig. 2. A safety violation in the implementation of PBFT in [Cao. 2020].

They ensure the liveness property of *termination* under the assumption of a (partially) synchronous network [Dwork et al. 1988] to overcome the FLP impossibility result [Fischer et al. 1985] for asynchronous networks. In this work, we check for the safety properties of validity, integrity, and agreement, as well as the liveness properties of termination and the completion of a client request.

3 OVERVIEW

In this section, we demonstrate how ByzzFuzz discovers bugs using example violations that we detected in our empirical evaluation. We describe (i) two executions of PBFT [Castro and Liskov 1999], one of which exposes a bug in the PBFT protocol and another one that detects a bug in an implementation of PBFT, and (ii) an execution of the XRP Ledger Consensus Algorithm [Schwartz et al. 2014] that uncovers a previously unknown bug in the production implementation of Ripple.

Our first example serves to show that ByzzFuzz can detect bugs using a combination of network and process faults, the second example showcases the detection of a safety violation of agreement, and the third one showcases the detection of a liveness violation of termination in the existence of a single process fault. Apart from the omission faults in the first example, both agreement and liveness violations manifest under process faults that corrupt the message fields with small perturbations to their original values, which we refer to as *small-scope mutations*.

For each example, we provide the fault configuration randomly generated by ByzzFuzz that detects the bug, and then we describe the execution with the violation.

3.1 Example Violations of Consensus in PBFT

Violation of liveness in the PBFT Algorithm. ByzzFuzz produced the violation of liveness given in Figure 1b, using two different fault configurations. The produced executions realize a known violation described in [Berger et al. 2021]. The violation prevents the client process from completing its request under the read-only optimization.

ByzzFuzz fault configurations. In the first test execution that detects the violation, ByzzFuzz injects two process omission faults into the randomly selected rounds, PRE-PREPARE and REPLY, respectively. The omissions prevent the PRE-PREPARE message from being delivered to $p3$ in the

first round and the REPLY message from being delivered to the client in the reply round. In the second one, ByzzFuzz randomly generates a network partition fault and a process fault. It injects the network partition fault to isolate the randomly selected process $p3$ from the rest of the cluster in the PRE-PREPARE round and injects a process fault to omit or corrupt the REPLY message.

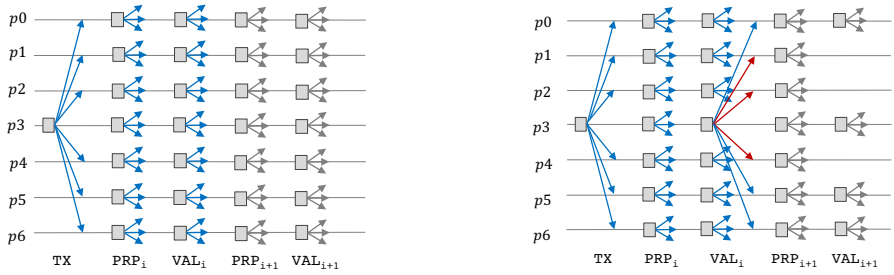
The buggy execution. Figure 1b illustrates the violation showing the omitted or lost messages by dotted red arrows. In the PRE-PREPARE round, $p3$ does not receive the PRE-PREPARE message to process a request; therefore, it does not participate in the rest of the execution. However, $p2$ and $p3$ progress in the following rounds by collecting $2f + 1$ PREPARE messages followed by $2f + 1$ COMMIT messages. After they commit, $p2$ and $p3$ send REPLY to the client. However, the faulty process $p0$ omits the reply and prevents the client from receiving $f + 1$ matching replies and completing the request. This results in a failed request that cannot be resolved by a synchronous network ($p0$ can repeatedly omit the reply) or by a view change request ($p0$ can block a view-change as it is active to the other processes).

Violation of agreement in the PBFT implementation. ByzzFuzz discovered a bug in the implementation of PBFT in [Cao. 2020] by injecting a single randomly generated Byzantine process fault into the execution. The bug causes the processes to fail to agree on their logs of executed client operations.

ByzzFuzz fault configuration. ByzzFuzz detected the bug by injecting a single Byzantine process fault with a *small-scope message mutation* into a single round of the protocol execution. ByzzFuzz randomly selects the faulty process, the subset of processes to receive faulty messages, and the protocol round to inject the fault. In the execution that triggers the violation, the faulty process $p0$ sends an incorrect PRE-PREPARE message to $p3$. Instead of $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle, m\rangle$, it sends $\langle\langle \text{PRE-PREPARE}, v, \mathbf{n+1}, d\rangle, m\rangle$ with an incorrectly incremented sequence id.

The buggy execution. Figure 2 illustrates the execution, which processes two client requests $r1$ and $r2$. We mark the sequence number and the client operation processed in a message as a tuple $\langle v, n\rangle$. Upon receiving $r1$, $p0$ assigns the sequence number $n = 0$ to $r1$ and multicasts $\langle\langle \text{PRE-PREPARE}, v = 1, n = 0, d\rangle, r1\rangle$. ByzzFuzz injects a process fault in this round and applies a small-scope mutation to the message sent to $p3$. Instead of the correct message, it sends the message to $p3$ with an incremented sequence number (shown by the red arrow). The process $p3$, which has not accepted another proposal for $n = 1$, accepts the proposal. While the others send PREPARE messages for $\langle 0, r1\rangle$, it incorrectly prepares for $\langle 1, r1\rangle$. However, it does not receive any matching PREPARE messages and does not progress. Meanwhile, the other processes accept and commit the operation requested at $r1$ at the sequence number $n = 0$. The client also completes the request since it receives at least $f + 1$ matching replies. Upon receiving the second request $r2$, $p0$ sends PRE-PREPARE message for $r2$ with the sequence number $n = 1$. This time, all processes, including $p3$, progress for $n = 1$ and commit the operation requested at $r2$. However, $p3$ commits the operation of $r1$ at $n = 1$, while the other processes commit the operation of $r2$ at $n = 1$. The inconsistency of the operation logs violates the agreement property of consensus. The violation is caused by an incorrect implementation of the protocol, which handles the PRE-PREPARE messages incorrectly.

Note that the bug exposes by modeling process faults using *small-scope mutations*, *i.e.*, applying small perturbations to the original message. An execution corrupting the message content with an arbitrary sequence number does not produce the violation since (i) the messages are discarded by the processes if its sequence number is not in an expected interval, and (ii) even if it is the expected interval, the bug does not manifest if the incorrect sequence id does not coincide with the sequence id of a successful subsequent request.



(a) The normal case execution.

(b) A newly discovered violation of termination.

Fig. 3. The normal case execution and an execution violating termination of the XRP LCP of Ripple.

3.2 Example Violations of Consensus in Ripple

Protocol description. The XRP Ledger Consensus Protocol of Ripple (XRP LCP) [Schwartz et al. 2014] is developed for the Ripple XRP Ledger, a blockchain-based distributed ledger payment system. The protocol ensures agreement of the processes (also called validators) on the order of blocks of transactions to be appended to the XRP Ledger in an adversarial environment. The protocol tolerates up to f Byzantine processes in a cluster of $5f + 1$ trusted processes.

Figure 3a illustrates the normal case execution of the XRP Ledger consensus protocol. The processes start in the open phase, where they receive transactions from the clients and disseminate them to the other processes (illustrated by the blue arrows in the TX round in Figure 3). Then, the processes run a PROPOSAL and a VALIDATION round to establish consensus on the set of transactions to commit in the next ledger.

The figure shows the execution of the protocol rounds for committing ledger i and $i + 1$. In the PROPOSAL round, the processes send $\langle \texttt{PROPOSAL}, txs, i, pre \rangle$ containing the set of transactions txs they received, the round number of the proposal, and the hash of the last fully validated ledger, respectively. In Figure 3a, all processes propose a ledger containing the blue transaction for the ledger i. The processes move to the VALIDATION round if they reach an agreement with 80% of the trusted processes or a predefined duration of time passes. In the VALIDATION round, the processes close the current ledger, which contains the transaction. They broadcast $\langle \texttt{VALIDATION}, id, txs, i, pre, t \rangle$ containing the ledger hash ($hash$), ledger sequence number (i), the set of transactions to be included (txs), and the hash of the last fully validated ledger ($prev$). If a process receives matching validations from at least 80% of its trusted processes, it fully validates the ledger and commits the transactions. Otherwise, it moves to the next open phase to collect transactions for the next ledger. In the normal case execution in Figure 3a, all processes agree on the ledger i containing the requested transaction (shown by the blue arrows). In the later protocol rounds, the processes do not receive any client transactions, and they vote for and validate an empty ledger for the sequence id $i + 1$ (shown by the grey arrows).

Violation of termination in the XRP LCP Algorithm. ByzzFuzz discovered a previously unknown bug in the implementation of Ripple that manifests as a violation of termination.

ByzzFuzz fault configuration. ByzzFuzz detects the bug by injecting a single process fault with a small-scope mutation in a single protocol round. Figure 3b illustrates the process fault and the buggy execution. The faulty process $p3$ sends an incorrect message to a subset of processes $\{p1, p2, p4\}$ in the $\texttt{VALIDATION}_\texttt{i}$ round for validating a ledger (shown by the red arrows). The injected process fault applies a small-scope mutation on the sequence id of the ledger in the VALIDATION message. While

$p3$ sends the correct message $\langle \texttt{VALIDATION}, id, txs, i, pre, t\rangle$ to the processes $\{p0, p5, p6\}$, it sends the incorrect message $\langle \texttt{VALIDATION}, id, txs, \mathbf{i+1}, pre, t\rangle$ to $\{p1, p2, p4\}$. Note that the bug exposes using a small-scope message mutation but not in an arbitrary mutation of the message fields.

The buggy execution. The Ripple processes receiving an incorrect VALIDATION message in the $\texttt{VALIDATION}_{\texttt{i}}$ round transition into an invalid internal state, which prevents these processes from making further progress. Upon receiving the message, the processes compare the current ledger in the message with their local information in $mLastValidLedger$, which stores the last validated ledger. The processes receiving the incorrect message have a mismatching sequence id with the ledger for which they already have seen a quorum of 80%. The bug occurs due to insufficient validation of the contents of incoming VALIDATION messages in the handling of VALIDATION messages. We provide more information about the bug in Section 5.3.

Violation of agreement in the XRP LCP Algorithm. ByzzFuzz discovered two different executions that created a fork in the cluster by injecting two process faults into the execution of the protocol. While one of the executions materializes a known vulnerability [Amores-Sesar et al. 2020], the other one uncovers a previously unknown scenario that violates agreement. We describe the violations we detected in our experimental work in Section 5.3.

4 THE BYZZFUZZ TESTING ALGORITHM

ByzzFuzz tests the fault tolerance of BFT consensus implementations by injecting randomly generated network and process faults into the runtime of the protocol execution. Similar to the state-of-the-art automated test generation tools for distributed systems (e.g., Jepsen), it models network faults using network partitions. ByzzFuzz differs from the existing tools by its generation of Byzantine process faults. More specifically, it models Byzantine process faults using structure-preserving *small-scope* message mutations.

ByzzFuzz inputs and outputs. The input to the ByzzFuzz algorithm is a set of fault-injection parameters along with a set of mutators for protocol messages. The fault-injection parameters are the number of protocol rounds with process faults, the number of protocol rounds with network faults, and the number of protocol rounds among which the faults will be injected. The mutators for protocol messages are accessed through an interface implementation that provides the mutation methods for the protocol messages under test. As output, ByzzFuzz provides the execution trace along with a log stating whether the execution trace violates consensus properties. During the execution with the injection of randomly generated faults, ByzzFuzz records the delivered, dropped, or mutated messages into an execution trace. At the end of the execution, it checks for the consensus properties on the collected execution trace. It reports the violations to the user in a log file listing the delivered, dropped, or mutated messages and stating the violated consensus properties, if any.

The algorithm utilizes the round-based communication structure of consensus protocols to introduce the network and process faults. The round-based communication of the processes [Charron-Bost and Schiper 2009; Elrad and Francez 1982; Moses and Rajsbaum 2002] provide a useful abstraction for analyzing the behaviors of consensus protocols [Aminof et al. 2018; Bertrand et al. 2019; Damian et al. 2019; Dragoi et al. 2020, 2014; Tsuchiya and Schiper 2011]. The executions of consensus protocols are organized as a sequence of lock-step *communication rounds* in which processes exchange messages following a set of protocol rules. In every round, the processes can *(i)* send messages to the other processes, *(ii)* receive and process the messages delivered in that round, and *(iii)* update their local states accordingly. The protocol rules define the new state of a process and which messages it will send based on its state at the beginning of the round and the messages it receives in the current round. For example, a normal case execution of the PBFT protocol consists of PRE-PREPARE, PREPARE, COMMIT, and REPLY rounds.

Exploiting the round-based execution structure enables protocol logic-aware injection of network and process faults. Consider a testing algorithm that drops or mutates a particular message sent to a process. Typically, the retry mechanism of the protocol implementation resends the same message until it gets a receipt acknowledgement or reaches a maximum number of trials. However, it is unlikely for a naive fault injection algorithm to drop or mutate all retransmissions of the same message. Round-based fault injection drops all retransmissions of the same message *that are sent in the same protocol round*. Similarly, it allows applying the same process fault to the messages delivered in the same round (e.g., omitting all messages or applying the same mutation to the messages delivered to some processes in that round). Delivery of the same incorrect content to a subset of processes can lead to a split-brain scenario, which can be targeted by a faulty process.

Second, sampling faults using a round-based algorithm makes it easier to reproduce a test case execution. While a test execution produced by ByzzFuzz involves some nondeterminism in the timing of the message delivery and, therefore, it is not completely deterministic, it injects the same faults into the same logical protocol rounds across different executions of the same test case.

Finally, round-based injection of faults significantly reduces the sample set of executions [Dragoi et al. 2020]. Instead of considering each individual message as a decision point to inject a fault, ByzzFuzz considers the protocol rounds as decision points and chooses to corrupt a message based on its logical protocol round, which is much smaller in number than the messages in an execution. Moreover, round-based injection of faults produces structured test executions that are easier to understand and debug [Dragoi et al. 2020].

ByzzFuzz further reduces the sample set of test executions by bounding the number of network and process faults in an execution. The bounded testing approach is based on the insight that complex executions that trigger bugs can be minimized to expose the bug within a small bound of some notion. For example, many bugs in concurrent programs expose in a few context switches [Musuvathi and Qadeer 2007], a few deviations from a deterministic scheduler [Emmi et al. 2011], or a few ordering constraints [Burckhardt et al. 2010; Kulahcioglu Ozkan et al. 2018; Niksic 2019]. ByzzFuzz adopts bounded testing to limit the exploration of executions with a few network or process faults. Bounding faults also helps in eliminating test executions that cannot make meaningful progress but execute repetitive retrial mechanisms due to frequent faults.

The ByzzFuzz algorithm distributes a parametrized number of network and process faults into the execution of the protocol rounds. In the next subsections, we describe how ByzzFuzz models the network and process faults and the algorithm for sampling and injecting these faults.

4.1 Modeling Network Faults

ByzzFuzz tests the behavior of consensus systems under network partition faults [Gilbert and Lynch 2002] that cause network splits between the distributed processes due to the failure of network devices. Network partitioning splits a cluster into multiple blocks so that a process in a block cannot communicate with the processes in the other blocks. Theoretical analysis of the space of executions with network faults [Majumdar and Niksic 2018] shows that a small set of randomly generated network partitions (e.g., by Jepsen [Kingsbury. 2022]) can provide a full coverage of behaviors empirically correlated with fault tolerance bugs.

We use set partitions to model network partition faults and sample from the partitions of a set uniformly at random. A partition of a set S is a collection $B_1, \ldots, B_k$ of nonempty disjoint subsets of S such that $B_1 \cup \ldots \cup B_k = S$ [Mansour 2013]. A partition of a set of processes $\mathbb{P}$ into a number of blocks models a network partitioning such that only the processes in the same block can communicate with each other while they are isolated from the processes in the other blocks.

Example 1. *The five possible partitions of the set of processes* $\{p0, p1, p2\}$ *are given by* $\{\{p0, p1, p2\}\}$, $\{\{p0, p1\}, \{p2\}\}$, $\{\{p0, p2\}, \{p1\}\}$, $\{\{p0\}, \{p1, p2\}\}$, *and* $\{\{p0\}, \{p1\}, \{p2\}\}$. *The first network partition is equivalent to a non-faulty network where all the messages between all pairs of processes are delivered. The last partition isolates all processes from each other, disabling the delivery of messages. Consider the second partition as an example. It isolates* $p2$, *while* $p0$ *and* $p1$ *can communicate with each other. Partitioning the network with this configuration will disable the messages sent from or to* $p2$ *and the other processes can form a quorum to progress in the protocol execution. A similar scenario happens in Figure 1b, where the partition in the first round isolates* $p3$ *and causes it to lag behind.*

4.2 Modeling Process Faults

ByzzFuzz bounds the scope of arbitrary process faults using two key approaches to produce faulty messages: First, we consider *structure-aware mutations* that only produce messages that are syntactically valid. Second, we apply *small-scope mutations* that mutate the messages by modifying the content of the correct messages with small perturbations.

Structure-aware mutations. Our algorithm injects arbitrary process faults into an execution by omitting or mutating messages sent from a faulty process. A straightforward method for mutating the messages uses random bit-level mutations to their serialized representations. However, such mutations will only exercise the message deserialization or parsing logic and most likely be discarded by the receiver processes.

Inspired by the high-level mutations used in input fuzz testing [Padhye et al. 2019; Park et al. 2020; Pham et al. 2020, 2021], we corrupt the messages sent by a faulty process by applying *structure-aware* high-level mutations to the messages. As opposed to random bit-level mutations, structure-aware mutations preserve the valid structure of the protocol messages. Hence, they can go deep into the protocol execution logic instead of only exercising the message parsing logic, and they can test the behavior of processes when they receive incorrect messages.

Our algorithm uses a set of structure-aware mutations that modify the values of the message fields as provided by the programmer. The algorithm mutates a message by omitting it or applying a randomly chosen structure-aware mutation among the set of mutations provided by the programmer. We give the set of structure-aware mutations we used for testing PBFT, Tendermint, and Ripple in Table 1, Figure 4a and Table 4 in Section 5, respectively.

Small-scope mutations. A challenge in mutating the value in a message field is the enormous set of its possible valuations. Instead of mutating a message field by replacing its value with an arbitrary one, we apply *small-scope mutations*. These mutations modify the value of a field with a small deviation from either its original value, *e.g.*, increasing or decreasing the value of a number field up to a small value, or in time, *e.g.*, by repeating a proposal value from a previous message.

Besides reducing the sample set of mutations, small-scope deviations to the original values of the message fields test the boundary values of the fields, making the mutated values more likely to collide with the previous or subsequent rounds of the protocol. In our implementation and evaluation, we mutate number-valued message fields in value, by only incrementing or decrementing them. For hash-valued fields, we use either a small-scope mutation in value (i.e., the hash of a transaction that applies a small-scope mutation to the original transaction) or in time (i.e., the hash of a transaction executed in the previous round).

Example 2. *In the execution in Figure 2, we mutate the sequence number of the* PRE-PREPARE *message by incrementing its value. This small-scope mutation is sufficient to expose the error in the message handling, which causes the processes not to check whether the proposal they promised for a sequence number matches with the proposal they prepare and commit later.*

Note that the bug manifests as an agreement violation using small-scope mutations but not necessarily by replacing the sequence number with some arbitrary value. The violation occurs in an execution where a process receives a proposal with an incorrect sequence number, and later in the execution, the processes decide to commit another proposal for the same sequence number. The problem does not manifest in the executions where (i) the mutated sequence number is not in a predefined watermark window which causes the PRE-PREPARE *message to be discarded, or (ii) the mutated sequence number is not used later for committing another proposal.*

Similarly, in the next example in Figure 3b, we mutate the sequence number of the VALIDATION *message by simply incrementing its value. This small-scope mutation exposes the error in the message handling, which causes the receiving process to move to an inconsistent state.*

4.3 The Testing Algorithm

ByzzFuzz injects network partitions and arbitrary process faults into an execution with a set $\mathbb{P}$ of processes that execute a given set of client requests.

Algorithm 1 shows how ByzzFuzz randomly samples network and process faults and injects them into an execution. It takes three input parameters: c, a bound on the number of protocol rounds in which a faulty process sends mutated messages, d, a bound on the number of network partition faults, and r, a bound on the number of rounds, among which the faults will be distributed.

The algorithm keeps a network partition fault as a pair $\langle round, partition \rangle$ that corresponds to the round number and a set partition of $\mathbb{P}$, modeling the network partition. The algorithm simulates the network fault by dropping the messages exchanged between the isolated processes. Similarly, it keeps the process faults as a triple $\langle round, procs, seed \rangle$ where $round$ is the round number in which the faulty process sends mutated messages (or omits them), $procs$ is the set of receivers of the mutated messages, and $seed$ is the random seed to determine how to mutate the message. The algorithm mutates a message of a faulty process by *(i)* randomly choosing a mutation using $seed$ among the mutations provided by the programmer for that message type, and *(ii)* mutating the messages delivered to $procs$ using the selected mutation.

For randomly sampling network and process faults, we use the following auxiliary functions:

- randomElementFrom: Chooses an element from the given set uniformly at random.
- randomPartitionOf: Chooses a partition of the given set uniformly at random. We sample from a uniform distribution of all possible partitions of the set of processes $\mathbb{P}$ by sampling from the precomputed set of partitions for small-sized clusters and using the urn model sampling algorithm [Stam 1983] for larger sets.
- randomSubsetOf: Chooses a subset of the given set uniformly at random.

In the onInit procedure, the algorithm randomly samples d network partition faults (line 3) by randomly selecting a round from r rounds and a partition from the set of partitions of $\mathbb{P}$. Then, it randomly chooses f processes to be faulty and randomly samples which messages from the faulty processes will be mutated.[3] For each of the c rounds with process faults, the algorithm randomly chooses a round in which the mutation occurs, the subset of processes that receive the mutated message, and the random seed upon which the mutation is based.

Intercepting Messages. When a process sends a message, the onMessage procedure is called with the message-in-flight as a parameter. Our algorithm uses the helper functions $\mathsf{rnd}(m)$, $\mathsf{sender}(m)$, and $\mathsf{recv}(m)$, which return the round number, the sender process, and the receiver process of the message m, respectively, and use this information to decide whether to drop or mutate a message. A round of a protocol is identified by protocol-specific metadata that is contained in the protocol messages (*e.g.*, PBFT protocol messages contain the view number, sequence number, and

[3]For simplicity, we present the algorithm for $f = 1$ faulty process, but it is easy to extend it for multiple faulty processes.

Input: A bound c on the #rounds with process faults
Input: A bound d on the #rounds with network faults
Input: A bound r on the #rounds with faults

`/* round, and a partition of` $\mathbb{P}$ `*/`
Data: *networkFaults*: Set[(Int, Set[Set[$\mathbb{P}$]])]
`/* round, a subset of` $\mathbb{P}$`, and a seed */`
Data: *procFaults*: Set[(Int, Set[$\mathbb{P}$], Int)]

1 **Procedure** onInit()
  `/* sample network faults */`
2 $networkFaults \leftarrow \emptyset$
3 **for** $i := 1$ *to* d **do**
4 $round \leftarrow$ randomElementFrom($[1, r]$)
5 $partition \leftarrow$ randomPartitionOf($\mathbb{P}$)
6 $networkFaults$ += $(round, partition)$
  `/* sample process faults */`
7 $p_{byz} \leftarrow$ randomElementFrom($\mathbb{P}$)
8 $procFaults \leftarrow \emptyset$
9 **for** $i := 1$ *to* c **do**
10 $round \leftarrow$ randomElementFrom($[1, r]$)
11 $procs \leftarrow$ randomSubsetOf($\mathbb{P}$)
12 $seed \leftarrow$ randomElementFrom($\mathbb{Z}$)
13 $procFaults$ += $(round, procs, seed)$
14 **Procedure** onMessage(m)
15 **if** (rnd*(m), π)* $\in$ *networkFaults* **and** isolates*(π,* sender*(m),* recv*(m))* **then**
  `/* do nothing, drop the message */`
16 **else if** sender(m) = p_{byz} **and** ((rnd(m), recv(m) $\cup$ _, $seed$) $\in$ $procFaults$) **then**
  `/* mutate and send the message */`
17 $M \leftarrow$ mutate($m, seed$)
18 send(recv(m), M)
19 **else**
20 send(recv(m), m)

Algorithm 1: Random injection of c process faults and d network partition faults into r rounds of protocol execution.

the protocol verb of their rounds). We associate the round of a message with the round of its sender, *i.e.*, the maximal round in which its sender sent or received a message. Hence, we apply the same faults to the retransmissions of a message *that are sent in the same protocol round*, but allow their nonfaulty delivery if they are repeated in a later round to synchronize the lagging processes.

The algorithm checks whether the message will be dropped by checking whether the sender and the receiver processes are isolated by a network partition in that round (line 15). If the message is not dropped, the algorithm checks whether the message will be mutated (line 16). For this, it checks whether the message belongs to a round with process faults, the sender of the message is a faulty process, and the receiver is in the set of receivers of the mutated messages. If this is the case, the algorithm mutates the message by applying a randomly selected mutation among a predefined set of mutations (line 17) and delivers the mutated messages (or an empty set of messages for omission) to the receivers (line 18).

Global vs. Subjective Trust. Traditional consensus protocols make a global, symmetric trust assumption, *i.e.*, the processes trust all other processes in the cluster and run consensus rounds to form a quorum. For these algorithms, ByzzFuzz selects the subset of receivers of a mutated message from the set of all processes, $\mathbb{P}$. For simplicity, we write Algorithm 1 for this assumption. Different from the traditional assumption of global trust, some blockchain systems, such as Ripple [Schwartz et al. 2014] and Stellar [Lokhava et al. 2019], assume subjective, asymmetric trust [Cachin and Tackmann 2019] with an aim to offer open network membership. The processes in these systems do not necessarily trust all the processes in the cluster, but they can make their own subjective trust assumptions. That is, each process $p_0, \ldots, p_p$ in the cluster declares a list of processes they trust, i.e., $P_0 \subseteq \mathbb{P}, \ldots, P_p \subseteq \mathbb{P}$, respectively. Then, the processes $p_0, \ldots, p_p$ do not run the consensus rounds to form a quorum globally by $\mathbb{P}$, but by the subsets of processes they trust, i.e., $P_0 \subseteq \mathbb{P}, \ldots, P_p \subseteq \mathbb{P}$. For the consensus protocols with subjective trust, the testing algorithm first selects one of the sets P_i from $P_0, \ldots, P_p$, each of which is trusted to reach consensus, then samples a subset of P_i to send the mutated messages to.

4.4 Correctness Specification

We check the correctness of the test executions by using the correctness specification of consensus protocols, *i.e.*, *termination*, *validity*, *integrity*, and *agreement* (see Section 2.2). For termination, we check its bounded variant, *bounded termination*, considering an execution to violate it if the consensus is not reached (or a client request is not processed) within a bounded amount of time.

Most BFT consensus protocols guarantee safety for both synchronous and asynchronous networks (with arbitrary message delays) but guarantee termination in the existence of (partial) synchrony [Dwork et al. 1988]. We check the consensus properties assuming a partially synchronous network that eventually behaves synchronously. We simulate this network model by recovering all the network faults after some duration of time, allowing the system to deliver all the messages to their recipients. Note that during this recovery period, Byzantine processes may still mutate messages. After all client requests in the test harness are completed or when the execution time exceeds some predefined timeout duration, we check for the correctness properties on the execution trace of exchanged messages.

4.5 Implementation

ByzzFuzz can be implemented on top of a network interception layer, which intercepts the in-transit messages of the protocol execution and decides whether to drop or deliver messages with/without mutation. 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.

The major implementation effort for building ByzzFuzz is building the interception layer, i.e., instrumenting the system under test to redirect the protocol messages to an interception layer instead of delivering them to the recipients directly. The layer captures the in-transit messages, and it can control whether to lose them or deliver them to the recipients.[4]

Algorithm 1 can be implemented on top of the interception layer to enforce a given fault configuration in the system execution. The implementation of the algorithm requires background knowledge about the protocol under test to extract the round information. In particular, the function $\mathsf{rnd}(m)$ (on lines 15-16 in Algorithm 1) needs information about the protocol to compute the protocol

[4]In this work, we implemented the interception layer for PBFT and Ripple and used the Netrix framework [Nagendra 2022] available for Tendermint to intercept its messages.

round of a message based on the content of its fields (e.g., sequence number, protocol verb). Given the round, sender, and receiver of the message, the algorithm decides to drop a message, mutate it, or directly deliver it to its recipient.

The set of possible mutations is provided to the algorithm by implementing a mutator interface with a mutation method, parametrized by the message type and a random seed to select a mutation of that message. We provided the message mutation methods manually in our implementation; however, it can also be automated, i.e., to apply certain types of mutations to certain typed message fields. When the algorithm decides to mutate a message, it simply calls the interface method (on line 17 in Algorithm 1) with the in-transit protocol message and the random seed. Then, it delivers the mutated message to its recipient. For the fault injection framework to deserialize, mutate, and again serialize the faulty messages sent by the Byzantine processes, we provide the cryptographic keys of the processes to the fault injection layer.

For each test execution, we collect the execution trace of the protocol messages that are dropped, mutated, or delivered on the interception layer. We check for the consensus properties on the collected execution histories and report the traces and the violations to the user as the test output. Root causing of the reported violations requires manual analysis of the test outputs.

5 EMPIRICAL EVALUATION

We present an empirical evaluation of ByzzFuzz on the open-source production implementations of two popular blockchain systems, Tendermint and Ripple, available on GitHub in [Tendermint 2021], and [XRPLF 2021], respectively. As these implementations are well-tested and mature, we also apply ByzzFuzz to a simple implementation of PBFT we found on GitHub [Cao. 2020], which we expect to contain more implementation bugs. Our evaluation aims to answer the following research questions.

RQ1. Is ByzzFuzz effective at detecting Byzantine fault-tolerance bugs in consensus implementations?

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

RQ3. Are small-scope message corruptions sufficient to manifest Byzantine fault-tolerance bugs in consensus implementations?

Methodology. We address **RQ1** by testing the implementations of PBFT, Tendermint, and Ripple using different test parameters for the number of network and process faults. For each test configuration, we report the number of violations by checking bounded termination, validity, integrity, and agreement properties for consensus.

For **RQ2**, we implement a naive baseline network and process fault injection algorithm and compare the performance of ByzzFuzz to that of the baseline algorithm. The baseline algorithm extends the state-of-the-art network fault injection methods (such as Jepsen [Kingsbury. 2022]) with the injection of Byzantine process faults. Different from ByzzFuzz, the baseline algorithm arbitrarily injects network and process faults without the restriction to round-based structure-aware small-scope mutations.

In our evaluation, we use a baseline algorithm instead of an existing method as the baseline since there are no Byzantine process fault-injection tools available that apply to our target systems. To the best of our knowledge, the only existing work that generates test cases to detect Byzantine-fault tolerance bugs is Twins [Bano et al. 2021]. Twins is implemented for DiemBFT, and it is available in the production repository of Diem [Diem 2021]. Our evaluation does not empirically evaluate Twins since its implementation is tightly bound to the implementation of the system under test [Diem 2021], and it is not available for Tendermint, Ripple, or PBFT. Porting Twins would likely require

Table 1. Structure-aware corruptions used for PBFT. Mutated values are primed and typeset bold.

Message	Mutations
$\langle\langle \texttt{PRE-PREPARE}, v, n, d\rangle, m\rangle$	$\langle\langle \texttt{PRE-PREPARE}, \mathbf{v'}, n, d\rangle, m\rangle$
	$\langle\langle \texttt{PRE-PREPARE}, v, \mathbf{n'}, d\rangle, m\rangle$
	$\langle\langle \texttt{PRE-PREPARE}, v, n, d\rangle\rangle, \mathbf{m'}$
$\langle \texttt{PREPARE}, v, n, d, i\rangle$	$\langle \texttt{PREPARE}, \mathbf{v'}, n, d, i\rangle$
	$\langle \texttt{PREPARE}, v, \mathbf{n'}, d, i\rangle$
$\langle \texttt{COMMIT}, v, n, d, i\rangle$	$\langle \texttt{COMMIT}, \mathbf{v'}, n, d, i\rangle$
	$\langle \texttt{COMMIT}, v, \mathbf{n'}, d, i\rangle$
$\langle \texttt{VIEW-CHANGE}, v, n, C, P, i\rangle$	$\langle \texttt{VIEW-CHANGE}, \mathbf{v'}, n, C, P, i\rangle$
	$\langle \texttt{VIEW-CHANGE}, v, \mathbf{n'}, C, P, i\rangle$
$\langle \texttt{NEW-VIEW}, v, n, C, P, i\rangle$	$\langle \texttt{NEW-VIEW}, \mathbf{v'}, n, C, P, i\rangle$
	$\langle \texttt{NEW-VIEW}, v, \mathbf{n'}, C, P, i\rangle$

a full reimplementation which is outside the scope of this paper. While we do not empirically evaluate Twins, we discuss how the test generation of Twins compares to ByzzFuzz and which of the bugs ByzzFuzz discovered cannot be detected by Twins in Section 6.

For **RQ3**, we compare the effectiveness of small-scope mutations, *i.e.*, mutations that apply small deviations to the original message field values, to any-scope mutations *i.e.*, mutations that modify the value of a message field arbitrarily. We implement any-scope variants of the message mutation methods for PBFT, Tendermint, and Ripple. We then compare the bug detection performances of the two approaches.

5.1 Testing a PBFT Implementation

To check the effectiveness of ByzzFuzz for detecting fault tolerance bugs, we started by testing an implementation of the PBFT protocol we found on GitHub [Cao. 2020]. It is a simple implementation of the protocol logic that avoids the complexities of a production implementation. We expected it to be less well-tested than a production system and likely to contain implementation bugs. We implemented ByzzFuzz for testing PBFT[5] by intercepting the protocol messages in the execution and injecting faults randomly generated by Algorithm 1.

We tested the system running a cluster of four processes with a varying number of $c = [0, 2]$ rounds with process faults and a varying number of $d = [0, 2]$ rounds with network partitions distributed among $r = 8$ rounds in the execution.

We modeled Byzantine process faults for PBFT using the structure-aware mutations in Table 1.[6] We mutated the view number (v), sequence number (n), or the requested transaction (m). Our mutations do not modify the digest (d) since the computation of digests is omitted in the tested implementation. We implemented the mutations with both (1) small-scope mutations to the values, *e.g.*, by either incrementing or decrementing the numeric value fields, and (2) any-scope mutations that replace the original value of the message field with an arbitrary value. The algorithm randomly selects the faulty process and corrupts the messages from the faulty process using a randomly selected mutation in the randomly chosen rounds to the randomly chosen receiver processes.

Our evaluation detected several bugs in the PBFT implementation. Table 2 shows the test results for varying c rounds with process faults and d rounds with network faults. The fault configurations with $c = 0$ implies that we do not inject any small-scope or any-scope process faults. For these tests,

[5]The source code of ByzzFuzz for testing PBFT is available at https://github.com/burcuku/byzzfuzz-pbft.
[6]The mutations can be extended to duplicate or apply multiple mutations at once.

Table 2. Testing the PBFT implementation using c rounds with process faults and d rounds with network faults. For each test configuration, we report the detected number of termination (T), validity (V), integrity (I), and agreement (A) violations in the columns. For the tests with $c = 0$, we do not inject any small-scope or any-scope process faults and report the number of violations detected by only injecting d network faults. For the tests with $c > 0$, we inject either small-scope (ss) or any-scope (as) process faults and report the number of violations in the ss and as columns, respectively.

faults	**T**		**V**		**I**		**A**		**Total**	
baseline	41		0		0		0		41	
$c = 0,\ d = 1$	34		0		0		0		34	
$c = 0,\ d = 2$	53		0		0		0		53	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 1,\ d = 0$	1	1	4	4	0	0	2	2	4	4
$c = 1,\ d = 1$	32	30	2	2	0	0	4	2	36	31
$c = 1,\ d = 2$	58	57	2	2	0	0	3	4	61	61
$c = 2,\ d = 0$	3	3	6	6	0	0	4	4	7	7
$c = 2,\ d = 1$	35	41	6	6	0	0	4	1	40	45
$c = 2,\ d = 2$	53	66	3	3	0	0	5	3	59	69

we report the number of violations detected by only injecting d network faults. For $c > 0$, we tested the system with both small-scope (ss) or any-scope (as) process faults. For each configuration of the parameters, we repeated the tests for 200 times using different random seeds. The columns in Table 2 list the number of test executions that detect violations to bounded termination (T), validity (V), integrity (I), and agreement (A), respectively. Note that a test execution may produce multiple violations, e.g., the same execution can violate both agreement (where the processes diverge in their decisions for a user request) and bounded termination (where they fail to make an agreement for another user request). The last column lists the total number of executions with a violation.

Termination violations. We marked an execution as a violation of bounded termination if the majority of processes fail to process a client request in 20 seconds. The baseline test executions with termination violations are caused by crashing processes upon parsing errors for syntactically incorrect messages. The baseline algorithm arbitrarily corrupts the protocol messages, which are not properly handled in the simple implementation under test, which, in turn, crashes the processes in the cluster and prevents the processing of the operations. In our experiments, none of the termination violations generated by the baseline algorithm could detect the bugs in the implementation of the protocol logic that are detected by the structure-preserving message corruptions.

The violating executions generated by ByzzFuzz detected multiple bugs in the implementation of the protocol logic as well as the known protocol vulnerability described in Section 3, *i.e.*, which results in a liveness violation [Berger et al. 2021]. The violations due to implementation bugs are caused by an error in the assignment of sequence numbers and incorrect processing of prepared certificates. An example violation manifests in the existence of a network partition in a single protocol round, preventing some processes in the cluster from participating in the `PREPARE` round. The implementation error in the assignment of sequence numbers prevents the processes to reach an agreement in the later steps of the protocol. Another execution manifests the violation in a more complicated scenario where the processes running a `VIEW-CHANGE` after some network faults do not hear from the leader process and want to move to the next view with the new leader. In order for the new leader to complete the incomplete requests from previous rounds, the processes send the

certificates of the prepared requests in their VIEW-CHANGE messages. An implementation bug in the processing of the prepared certificates causes the processes to omit some prepared certificates in their VIEW-CHANGE messages and not successfully commit the requests with the missing certificates. ByzzFuzz detects that execution by injecting a network fault that leads to a VIEW-CHANGE round and injecting a process fault in the NEW-VIEW round. The bug manifests as a termination violation in which the cluster is prevented from agreeing on the new view and making progress, leading to an unresponsive state.

Validity violations. The validity violations are caused by an implementation error that omits the processing of message digests and incorrectly handles the PRE-PREPARE messages. When a process receives a PRE-PREPARE message, it does not check the validity of the client request m by comparing the message digest d to m. This causes the processes to accept incorrect proposals sent by the Byzantine leader. In the executions with a faulty leader that mutate the request m before sending it, the receivers accept an invalid request m instead of the original one sent by the client.

Agreement violations. The violations of agreement occur when processes decide on committing different values with a sequence number. Some agreement violations we detected expose in the execution scenario described in Section 3. Some other agreement violations we detected share the same root causes with some of the termination violations, *i.e.*, incorrect assignment of sequence numbers, and a missing prepared certificate in NEW-VIEW messages. In these executions, some processes fall behind in committing a request, while others commit the request but omit it in their NEW-VIEW messages. In that case, when a process missing a commit starts processing the message with a new PRE-PREPARE message, it commits the request with a different sequence number.

Other violations share the same root cause as the validity violations, *i.e.*, a missing check of the message digest that allows the Byzantine leader to send conflicting proposals. When a process receives a corrupted message m' from the leader at PRE-PREPARE and commits that value, while others receive and commit the correct version of m. In the executions where the majority of the processes commit the corrupted value, the cluster decides on the corrupted value m', processing a request that is different from the original client request.

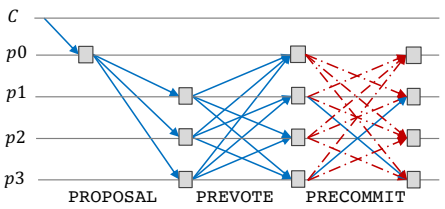
RQ1. Our experiments on testing PBFT show that ByzzFuzz can detect Byzantine fault tolerance bugs. ByzzFuzz can trigger violations that expose under certain fault configurations. In our tests, it could produce violations that manifest in the existence of (i) only network or message omission faults (as given in Figure 1b), or (ii) only process faults that mutate message contents (as given in Figure 2, or a combination of message omission and corruption (e.g., a test execution that triggers a termination violation mutates a PRE-PREPARE message and omits a REPLY message).

RQ2. We address research question **RQ2** using a baseline fault injection algorithm that drops or arbitrarily corrupts messages at random. As shown in Table 2, we could not detect any safety violations using the baseline fault injector. It could only produce some termination violations due to crashing processes upon receiving syntactically invalid protocol messages, which are not properly handled and crashed the processes in the simple implementation under test.

RQ3. To address **RQ3**, we repeated our tests using any-scope variants of the mutations in Table 1, which do not corrupt a message field value by applying a small deviation to its original value but replace the original message values with an arbitrary value. The columns titled with *ss* and *as* in Table 2 show the detected number of violations using small-scope and any-scope corruptions, respectively. In our evaluation, all the bugs detected by any-scope mutations are also detected by small-scope mutations. For the tests with $c = 1$ and $d = 2$, the higher number of agreement violations reported by *as* is caused by the inherent concurrency nondeterminism. While we did not observe any violations that were detected by any-scope corruptions but not detected by small-scope corruptions, we detected a wider set of violations using small-scope corruptions. For example, the

Message	Mutations
$\langle \texttt{PROPOSAL}, h, e, v, i \rangle$	$\langle \texttt{PROPOSAL}, h, \mathbf{e}', v, i \rangle$
	$\langle \texttt{PROPOSAL}, h, e, \mathbf{v}', i \rangle$
$\langle \texttt{PREVOTE}, h, e, d(v) \rangle$	$\langle \texttt{PREVOTE}, h, \mathbf{e}', d(v) \rangle$
	$\langle \texttt{PREVOTE}, h, e, \mathbf{d}(\mathbf{v})' \rangle$
$\langle \texttt{PRECOMMIT}, h, e, d(v) \rangle$	$\langle \texttt{PRECOMMIT}, h, \mathbf{e}', d(v) \rangle$
	$\langle \texttt{PRECOMMIT}, h, e, \mathbf{d}(\mathbf{v})' \rangle$

(a) Structure-aware mutations.



(b) A potential violation of termination.

Fig. 4. The structure-aware mutations and a potential violation of termination in Tendermint.

agreement violation in Figure 2 did not expose under the same configuration using any-scope corruptions. The violation manifests only when the incorrect sequence number in the corrupted message is used in some correct messages in later rounds.

5.2 Testing the Tendermint Consensus Algorithm

The Tendermint algorithm [Buchman 2016; Buchman et al. 2018] is a variant of PBFT that is designed to solve the consensus problem in the blockchain context and optimized for a high number of processes (also called *validators*). It reduces the complexity of message communication by relying on a peer-to-peer gossip protocol [Demers et al. 1987] among the processes.

The processes agree on committing a block of transactions by running a sequence of PROPOSAL, PREVOTE and PRECOMMIT rounds,[7] where we refer to a sequence of them as an *epoch*. The PROPOSAL of a block is followed by two voting rounds PREVOTE and PRECOMMIT. A block is committed when at least $2f + 1$ validators PREVOTE and then PRECOMMIT for the same block. The committed blocks are appended to a chain, with one block at each height.

We tested Tendermint v0.34.7 [Tendermint 2021] on a cluster of four nodes. Our implementation of ByzzFuzz intercepts the protocol messages using the Netrix Framework [Nagendra 2022] and injects faults randomly generated by the ByzzFuzz algorithm.[8]

Figure 4a lists the structure-aware mutations we used for testing Tendermint. The protocol message fields include the protocol round, the height of the blockchain (h), and the epoch number (e) for the current height. The fields v and $d(v)$ correspond to the proposal value and its hash, respectively. Our mutations modify the current epoch number e and the proposed value v. Our small-scope mutations increment the epoch number or set the proposal value, the hash value, to *nil*, which is sent in the rounds with an insufficient valid proposal or prevote messages received. For any-scope mutations, we set the epoch number to a random value and mutate $d(v)$ by assigning the hash of any other block.

Table 3 shows the number of violations we detected by running 200 tests for each configuration of test parameters in the ranges of $c = [0, 2]$, $d = [0, 2]$ and $r = 10$. We ran each of the tests for a maximum duration of one minute, followed by an additional minute of execution without any network faults (*i.e.*, delivering all messages). We checked for the validity, integrity, or agreement requirements using the commit logs of the processes. We marked an execution as a violation of bounded termination if the cluster fails to achieve agreement at the end of the execution.

While we could not detect any violations of safety, we detected several test executions that failed to terminate within a bounded amount of time. Figure 4b illustrates such a test execution that is

[7]Tendermint terminology uses *steps* instead of *rounds*.

[8]The source code of ByzzFuzz for testing Tendermint is available at https://github.com/wildarch/tendermint-byzzfuzz.

Table 3. Testing Tendermint using *small-scope* (ss) and *any-scope* (as) corruptions with varying d number of rounds with network partition faults and c rounds with process faults.

faults	**T**		**V**		**I**		**A**		**Total**	
baseline	45		0		0		0		45	
$c = 0,\ d = 1$	48		0		0		0		48	
$c = 0,\ d = 2$	94		0		0		0		94	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 1,\ d = 0$	0	0	0	0	0	0	0	0	0	0
$c = 1,\ d = 1$	53	67	0	0	0	0	0	0	53	67
$c = 2,\ d = 0$	0	0	0	0	0	0	0	0	0	0
$c = 2,\ d = 1$	38	69	0	0	0	0	0	0	38	69

produced by injecting a single network partition fault in the PRECOMMIT round of the execution. The network fault partitions the cluster into $\{\{p0\}, \{p1, p3\}, \{p2\}\}$ that only allows the communication of $p1$ and $p3$ in the PRECOMMIT round. In the test execution, the processes did not repeat their attempt to send the PRECOMMIT message and did not make progress after the network is healed. All violations to bounded termination we detected occur due to a similar case with a network partition. We do not encounter any violations with $d = 0$ network faults.

For **RQ2**, we compared ByzzFuzz to a baseline fault-injection algorithm that drops or corrupts messages arbitrarily at random. As given in Table 3, the baseline algorithm detects many violations of termination and shows comparable performance to ByzzFuzz in our Tendermint tests. Similar to the executions ByzzFuzz detected, the violations occur due to the processes not retrying their attempts when some messages are lost. For **RQ3**, we repeated our tests using any-scope mutations to messages. Our tests did not detect any violations of safety using either small-scope (*ss*) or any-scope (*as*) mutations.

Analysis of detected violations of termination. We discussed the violations of termination with Tendermint researchers and developers and concluded that the detected executions break the network assumptions of the Tendermint protocol. Tendermint relies on a gossip communication layer that implements reliable broadcast, which assumes eventual delivery of the messages even in the existence of network partitions. Therefore, it requires buffering the messages and resending them in the communication layer so that the messages can be delivered after the network recovers [Buchman et al. 2022, 2018]. The fault injection in our implementation does not buffer the messages sent during a network partition and does not ensure their delivery after the network heals. It only delivers messages that are sent after the network recovers. Hence, it does not satisfy the delivery guarantees of the gossip communication assumed by Tendermint. While the detected buggy executions cannot occur on a network that satisfies the assumed delivery guarantees, they can occur in real-world networks that may violate the assumptions (e.g., due to an overflow in the message buffer). Therefore, we report these executions as *potential* violations of termination.

5.3 Testing the XRP Ledger Consensus Protocol of Ripple

The XRP Ledger Consensus Protocol (XRP LCP) [Schwartz et al. 2014] is developed for the Ripple XRP Ledger (XRPL), a blockchain-based distributed payment system.

As explained in Section 3.2, the processes running the protocol receive and disseminate transactions from the clients, and run PROPOSAL and VALIDATION rounds for agreeing on the next ledger

Table 4. Structure-aware mutations used for Ripple.

Message	Mutations
$\langle$TRANSACTION, $s, r, amt\rangle$	$\langle$TRANSACTION, $s, r, \mathbf{amt'}\rangle$
$\langle$PROPOSAL, $txs, i, pre\rangle$	$\langle$PROPOSAL, $\mathbf{txs'}, i, pre\rangle$
	$\langle$PROPOSAL, $txs, \mathbf{i'}, pre\rangle$
$\langle$VALIDATION, $id, txs, i, pre, t\rangle$	$\langle$VALIDATION, $\mathbf{id'}, \mathbf{txs'}, i, pre, t\rangle$
	$\langle$VALIDATION, $id, txs, \mathbf{i'}, pre, t\rangle$

to append to the blockchain. Our implementation of BYZZFUZZ intercepts the protocol messages in the execution and injects faults randomly generated by the algorithm.[9]

Different from the traditional consortium-based consensus algorithms with a common global trust assumption for all validators, Ripple assumes subjective distributed trust [Cachin and Tackmann 2019; Cachin and Zanolini 2021]. The processes in the Ripple network declare the list of the validators they trust in a so-called Unique Node List (UNL) and only consider the proposals from their trusted set of validators for voting. While subjective trust brings some advantages, it makes the system more vulnerable to agreement violations that can cause forks in the network. The original design of the algorithm [Schwartz et al. 2014] states that the protocol ensures consensus given that the UNLs have a minimum 20% overlap in their trusted lists of validators. However, later analyses [Armknecht et al. 2015; Chase and MacBrough 2018] suggest stricter requirements on their overlap.

We tested Ripple v1.7.2 [XRPLF 2021] using a UNL configuration that meets the UNL overlap requirements specified in its whitepaper [Schwartz et al. 2014]. In particular, we used the configuration given in [Amores-Sesar et al. 2020] that is theoretically described to violate the agreement property under some fault configurations.

We ran Ripple in a network of seven processes $\{p0, p1, p2, p3, p4, p5, p6\}$, each of which has five of the processes in their UNL. The processes $\{p0, p1, p2\}$ trust $UNL_1 = \{p0, p1, p2, p3, p4\}$ and $\{p4, p5, p6\}$ trust $UNL_2 = \{p2, p3, p4, p5, p6\}$. This is a valid configuration w.r.t. the protocol description [Schwartz et al. 2014] since it has 60% overlap in its UNLs.

We modeled Byzantine process faults using the structure-aware mutations in Table 4. We mutated the amount of a transaction (amt), the set of transactions proposed or validated (txs), the identifying hash (id), and the sequence number (i). We implemented the small-scope mutations by incrementing the numeric fields and replacing a hash with the most recent value in the network. For any-scope mutations, we assigned a random value to a numeric field and replace a hash with one chosen at random from all of the previously encountered hashes in the execution. We simulated network partitions by withholding messages between the isolated processes until the network recovers. This models the reliability guarantees of the TCP channels in the overlay network assumed by Ripple implementation.

Table 5 lists the results of testing Ripple with $n = 7$ processes, injecting $c = [0, 2]$ process and $d = [0, 2]$ network faults distributed among $r = 6$ rounds. We ran 300 tests for each configuration.

BYZZFUZZ detected several executions that violate termination and two executions that violate agreement, including a previously unknown implementation bug that manifests as a termination violation, materializing the termination and agreement violations described in recent work [Amores-Sesar et al. 2020; Chase and MacBrough 2018], and a new variant of the described agreement violation

[9]The source code of BYZZFUZZ for testing Ripple is available at https://github.com/SERG-Delft/ConsensusTesting/tree/analyzer.

Table 5. Testing Ripple using *small-scope* (ss) and *any-scope* (as) mutations with varying d rounds with network partitions and c rounds with process faults.

faults	**T**		**V**		**I**		**A**		**Total**	
baseline	2		0		0		0		2	
$c = 0,\ d = 1$	11		0		0		0		11	
$c = 0,\ d = 2$	20		0		0		0		20	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 1,\ d = 0$	9	21	0	0	0	0	1	0	10	21
$c = 1,\ d = 1$	27	20	0	0	0	0	0	0	27	20
$c = 1,\ d = 2$	19	23	0	0	0	0	1	0	20	23
$c = 2,\ d = 0$	31	25	0	0	0	0	0	0	31	25

that results in a different fork in the cluster. The experimental results support the effectiveness of ByzzFuzz in detecting previously known and unknown bugs, addressing **RQ1**.

For **RQ2**, we tested Ripple with a baseline fault-injection algorithm that drops or corrupts messages arbitrarily at random. The baseline algorithm could only detect some termination violations, which share the root cause of the violation given in [Chase and MacBrough 2018]. However, it could not detect the termination violation presented in Section 3.2 or any agreement violations.

The experimental results support the effectiveness of small-scope mutations compared to any-scope mutations, addressing **RQ3**. The termination violation presented in Section 3.2 can only be detected using small-scope mutations of the sequence number but not with any-scope mutations.

Analysis of detected violations. We analyzed the test execution traces that violate termination or agreement and discovered two previously unknown violation scenarios as well as reinstating some known scenarios from [Amores-Sesar et al. 2020; Chase and MacBrough 2018]. We reported the two new violation scenarios to Ripple's developers, and they confirmed them. One of the violations is caused by an implementation error in the source code, whose fix will be included in version 1.10.0. The second violation is caused by an insufficient overlap in the UNLs. Since it can be avoided by a network configuration with a higher percentage of UNL overlap, it does not require any fix in the source code. In the rest of the section, we summarize four different violation scenarios ByzzFuzz detected in Ripple.

Violation of termination (1). This is the violation of termination explained in Section 3.2 that is caused by an implementation bug in Ripple's source code. The current protocol implementation processes the incoming `VALIDATION` messages based on the expected scenario with correct processes in mind. It does not sufficiently check the content of the message and mishandles it in case a Byzantine process sends a message with the sequence id of the next ledger.

The bug occurs when ByzzFuzz applies a process fault to the `VALIDATION` message, mutating $\langle \texttt{VALIDATION}, hash_i, txs, i, pre, t\rangle$ to $\langle \texttt{VALIDATION}, hash_i, txs, \mathbf{i+1}, pre, t\rangle$. When a process receives that message after reaching a quorum of 80% for ledger i, the checkAccept method (Algorithm 2) causes it to transition into an invalid internal state. This is because, although Ripple has a list of checks for the sequence number, the current implementation does not sufficiently check whether the provided hash $hash_i$ matches the sequence number $i + 1$. Therefore, it incorrectly considers the altered pair of hash ($hash_i$) and sequence number ($i + 1$) as the identifier of the last valid ledger, keeping it in the local variable $mLastValidLedger$ (see the checkAccept method in Algorithm 2).

Since the information in $mLastValidLedger$ is corrupted, the consistency check for the ledger in the received message fails. The correct processes send a `VALIDATION` message only for a ledger that

is a descendant of the last fully-validated ledger. This is ensured by the areCompatible method, which checks whether the current ledger (*view*) is compatible with the last fully-validated ledger stored in $mLastValidLedger$. The sequence number of the current ledger, $i + 1$, is equal to the sequence number of the corrupted ledger in $mLastValidLedger$, $i+1$. This is the consequence of the small-scope corruption incrementing i by only one. However, their hashes $hash_i$ and $hash_{i+1}$ are different, making the check fail. This causes the process to not participate further in the consensus algorithm and prevents the cluster from making further progress in the consensus.

The areCompatible method always returns `true` for sequence numbers below the one stored in $mLastValidLedger$. Hence, the bug will only manifest once the correct process reaches the mutated index from the Byzantine `VALIDATION` message. Small-scope corruptions expose the bug almost immediately, whereas with any-scope mutations, this can take an arbitrary amount of time.

We reported the bug to Ripple's developers, and they confirmed it, and the source code is fixed.[10] The fix correctly processes the incoming `VALIDATION` messages, preventing the Byzantine attack.

```
1  Procedure checkAccept(hash, seq)
2      validations ← validations(validators, hash)
3      if ⌈0.8 · |validators|⌉ ≤ |valCount| then /* check quorum for hash */
4          if seq > mLastValidLedger.seq then /* check if ledger is newer */
5              mLastValidLedger ← (hash, seq)
6  Procedure areCompatible(view, mLastValidLedger)
7      seq ← mLastValidLedger.seq
8      hash ← mLastValidLedger.hash
9      if seq = seq(view) ∧ hash ≠ hash(view) then
10         return false /* incompatible ledger */
11     else if seq < seq(view) ∧ hash ≠ ancestorHash(view, seq) then
12         return false /* incompatible following ledger */
13     return true
```

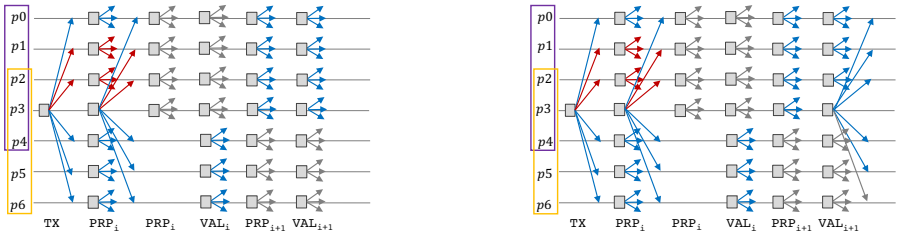
Algorithm 2: Excerpts from the Ripple source code. The method `checkAccept` checks if the specified ledger can become the new last fully-validated ledger ($mLastValidLedger$), and `areCompatible` checks whether the current view is compatible with $mLastValidLedger$.[12]

Violation of termination (2). ByzzFuzz detected another test scenario that causes a violation of termination in the existence of a single process fault. Unlike the previous termination violation, which can occur in any network configuration, this violation occurs when the processes in the network trust in different sets of processes (UNLs).

A test execution producing the violation is illustrated in Figure 5a. During the dissemination of a transaction T, the Byzantine process $p3$ corrupts the transaction it sends to the processes $\{p1, p2\}$. The processes trusting UNL_1 (shown by the purple box) do not achieve a majority on any transaction and close their ledger with an empty set of transactions (the grey arrows in $\mathsf{PRP_i}$). In the meantime, $\{p4, p5, p6\}$ close the ledger containing $\{T\}$. In the validation round, $\{p0, p1, p2\}$ try to validate the empty ledger (shown with grey arrows), and $\{p4, p5, p6\}$ try to validate the ledger with $\{T\}$ (shown with blue arrows). While UNL_1 reaches quorum and continues to make progress, the processes in UNL_2 get stuck. The execution materializes a scenario that is similar to the stuck network scenarios described in [Chase and MacBrough 2018] and [Amores-Sesar et al. 2020].

[10] https://github.com/XRPLF/rippled/pull/4424

[11] The pseudocode is based on the `checkAccept` and `areCompatible` methods in Ripple's source code.



(a) An execution with a violation of termination.

(b) An execution with a violation of agreement.

Fig. 5. Two executions of the XRP LCP of Ripple. The execution in (a) reproduces a known termination violation, and (b) uncovers a previously unknown variant of a known agreement violation.

We also detected a slightly different version of the execution in Figure 5a, where the divergence occurs due to a network fault. In this execution, the processes that close the ledger with an empty transaction do not end up with an empty one due to receiving a conflicting transaction but because they do not receive any transaction before closing their ledgers. The execution results in a similar divergence in process votes and causes a violation of termination. The baseline fault injection algorithm could also detect this violation in two test executions.

Violation of agreement (1). ByzzFuzz detected a violation of agreement that materializes a previously known attack scenario theoretically described in [Amores-Sesar et al. 2020]. Additionally, ByzzFuzz produces another violation in a similar test execution but one that results in a different cluster state. For brevity, we describe the new variant we detected.

Violation of agreement (2). We illustrate the new variant in Figure 5b. During the dissemination of the received transactions, the faulty process $p3$ sends an incorrect transaction to $\{p1, p2\}$ and proposes it in the subsequent round (red arrows, TX, PRP_i). The disagreement over $\{T\}$ and $\{T'\}$ in UNL_1 causes the processes to propose $\emptyset$ (grey arrows). $\{p0, p1, p2\}$ close and validate ledger i with no transactions after reaching 80% agreement, whereas $\{p4, p5, p6\}$ only close but not fully-validate ledger i containing $\{T\}$ (green arrows). UNL_1 proposes $\{T\}$ and UNL_2 proposes $\emptyset$ for the next ledger. UNL_1 closes and validates ledger $i + 1$ with $\{T\}$ due to the VALIDATION messages from $\{p0, p1, p2, p3\}$. The mutated VALIDATION message $p6$ receives from $p3$ allows $p6$ to see 80% agreement for its version of ledger $i + 1$, which does not contain any transactions. This creates a fork in the network, with processes diverging in their ledgers of committed transactions.

We can extend this scenario as shown in Figure 6. The extension allows an adversary to bypass Ripple's algorithm for detecting Byzantine behavior, which checks for conflicting VALIDATION messages. After the submission of T' and T to UNL_1 and UNL_2, respectively, $p3$ begins to show split behavior for proposals. Further, it alternates between agreeing with UNL_1 during even-indexed ledgers and UNL_2 during odd-indexed ledgers for validations. As sending a VALIDATION message for some ledger i entails implicit agreement with all of its preceding ledgers, it is sufficient for $p3$ to only agree to every other ledger and still ensure that the two forks make progress.

The variant differs from the violation [Amores-Sesar et al. 2020], as it allows the Byzantine process to go undetected. In the known violation scenario, the fork in the cluster is caused by committing two different ledgers, one with the correct transaction and the other with the mutated transaction. The Byzantine behavior will be detected by Ripple's algorithm, which checks for conflicting VALIDATION messages. In contrast, in the new violation, the Byzantine process does not send conflicting VALIDATION messages, and it will not get reported as Byzantine.

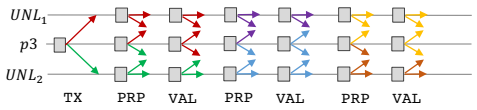


Fig. 6. An execution allowing skirting of the Byzantine behavior detection algorithm.

We reported the new violation scenario to Ripple's developers, and they confirmed it. The violation occurs due to insufficient overlap between the UNLs and can be avoided by a UNL configuration with a higher percentage of overlaps [Chase and MacBrough 2018]. While this vulnerability does not require a fix in the source code, the developers expressed interest in uncovering and analyzing such execution scenarios.

5.4 Summary of the Evaluation

We observe that ByzzFuzz is effective at detecting Byzantine fault-tolerance bugs in large-scale system implementations. To summarize, ByzzFuzz managed to discover several bugs that violate Byzantine fault tolerance of PBFT, Tendermint, Ripple:

- A liveness violation in the PBFT protocol which was described in [Berger et al. 2021].
- Three implementation bugs in the simple implementation of PBFT in [Cao. 2020] due to (1) incorrect assignment of sequence numbers in the protocol messages, (ii) incorrect processing of prepared certificates, and (iii) missing implementation of message digests.
- A potential protocol vulnerability in Tendermint, which is caused by not resending messages which can be lost due to network faults. We call it a *potential* violation since Tendermint assumes reliable delivery of the messages over gossip protocol, and the violation occurs only if the assumption is not satisfied by the underlying network.
- Two protocol vulnerabilities in Ripple previously described in [Amores-Sesar et al. 2020; Chase and MacBrough 2018] as well as a new variant of the agreement vulnerability. The vulnerabilities are caused by an insufficient overlap in the UNL configurations and can be avoided by network configurations with a higher ratio of overlap in the UNLs.
- A new implementation bug in the production code of Ripple, which insufficiently checks the message content before processing it, leading to a Byzantine fault-tolerance bug. The developers confirmed the bug, and the source code is fixed.

Except for the potential protocol vulnerability in Tendermint and a known protocol vulnerability in Ripple that causes a violation of termination, the bugs in the protocol logic could not be detected by the baseline fault injection algorithm.

Our evaluation also shows that ByzzFuzz's small-scope mutations are sufficient to expose all bugs. Moreover, some bugs are exposed only with small-scope and not arbitrary mutations. In particular, the violation in Figure 2 and a new violation in Figure 3b we discovered in Ripple expose using small-scope mutations. These bugs can be detected by any-scope mutations only with a lower likelihood. That is, provided that any-scope mutations form a valid message (e.g., with a sequence number in an expected watermark interval), they can trigger the bugs we found in a sufficiently long execution that reaches the processing of a block/ledger with the incorrect sequence number.

6 RELATED WORK

Although it has been decades since Byzantine fault-tolerant protocols were first proposed, some vulnerabilities have been discovered only recently and research on testing implementations is still

in its infancy. Except for a few works we will discuss, the existing techniques test system executions against concurrency, network faults, or benign process faults.

Several works focus on systematic exploration of distributed system executions [Deligiannis et al. 2016; Gunawi et al. 2011b; Killian et al. 2007; Leesatapornwongsa et al. 2014; Yang et al. 2009]. Systematic testing suffers from the state space explosion in the number of possible executions, and the existing methods combat this by employing partial order reduction [Godefroid 1996]. However, reduction to a feasible set of test executions typically requires some system-specific input to specify the system model or to incorporate nontrivial equivalence of system executions.

Alternatively, some existing tools use random or guided search techniques. Jepsen [Alvaro and Kingsbury 2020; Kingsbury. 2022] injects random process isolation or partition faults in the random points of system executions for checking consistency and isolation of distributed databases. Instead of injecting faults at random points, the work in [Dragoi et al. 2020] exploits the round-based protocol structure of the underlying executions for the injection of network faults. Following the fault injection as a service approach [Gunawi et al. 2011a], many systems [Bennett and Tseitlin 2012; Cotroneo et al. 2022; Hadoop. 2009; Izrailevsky and Tseitlin 2011; Kingsbury. 2022; Pogde et al. 2020; Rosenthal 2017] deployed random fault injection toolsets in their production systems.

Guided test generation approaches drive the test executions toward certain fault scenarios. PreFail [Joshi et al. 2011] programmable framework guides fault injection using probabilistic or heuristic approaches. Lineage-driven fault injection (LDFI) [Alvaro et al. 2016, 2015] uses the provenance graph of an execution to inject fault configurations that could prevent correct outcomes. CoFI [Chen et al. 2020] observes the current execution state and injects faults at the program points that are likely to exhibit buggy behavior. Crashtuner [Lu et al. 2019] crashes processes at specific program points where the system meta-info variables are accessed. FCatch [Liu et al. 2018] predicts time-of-fault bugs by observing conflicting operations in the correct executions of the system. Filibuster [Meiklejohn et al. 2021] injects faults into microservice architectures, the work in [Li et al. 2021] captures internal server nondeterminism in addition to network faults, and Sieve [Sun et al. 2022] injects faults into cluster management systems (*e.g.*, Kubernetes). Recent work [van Meerten et al. 2023] employs evolutionary algorithms to guide the concurrency test generation for consensus algorithms toward certain system behaviors. However, these works do not address Byzantine process faults and, therefore, cannot detect Byzantine fault-tolerance bugs.

Some existing work aims to improve the robustness of systems against Byzantine faults [Clement et al. 2009] or focus on performance attacks [Halalai et al. 2011; Lee et al. 2014; Martins et al. 2013; Singh et al. 2008]. The works in [Gupta et al. 2016] and [Dinh et al. 2017] present benchmark frameworks that evaluate BFT protocols and blockchains in various workloads and fault scenarios.

Recent works present fault injection frameworks for testing Byzantine fault-tolerant systems. Netrix [Nagendra 2022] and ZERMIA [Soares et al. 2021] allow developers to write test cases implementing tailor-made faults and testing systems with user-provided execution scenarios.

Closest to our work is a recent method, Twins [Bano et al. 2021], an automated test generator with Byzantine faults. Twins simulates Byzantine behavior by running multiple instances of a process with the same identity (called *twins*). Twin replicas emulate Byzantine behaviors that were previously observed in known attacks and exhibit three types of faulty behaviors: message equivocation (*e.g.*, sending different proposals to different recipients), double voting, and losing the internal process state. Twins systematically enumerates and exercises attack scenarios using twin replicas. In contrast to Twins, which targets adversarial behavior observed in *known attacks*, ByzzFuzz tests the system by sampling from the entire space of possible process faults (bounded by the programmer input for message mutations). This enables ByzzFuzz to cover a broader set of faulty processes in addition to equivocation, double voting, and losing the internal state.

Our empirical evaluation does not compare to Twins since it is only available for HotStuff [GitHub 2020, 2022] and DiemBFT [Diem 2021], but not available for PBFT, Tendermint, and Ripple. However, theoretically, Twins cannot detect some violating executions that ByzzFuzz detected. For example, Twins cannot detect the faulty executions in Figure 2 or in Figure 3, which uncovers a new bug in Ripple. These executions require a process fault that corrupts the sequence number of a message with the sequence number of a *future* round, which cannot be produced by a twin replica. While Twins allows replicas to lag behind by losing some process state, it does not inject faulty messages with the future values of protocol message fields. We believe ByzzFuzz and Twins can be used complementarily, bringing together systematic testing of Twin's set of adversary behaviors and ByzzFuzz's sampling from a broader set of faulty process behaviors.

7 CONCLUSION

This paper introduces ByzzFuzz, a new method for finding implementation errors in Byzantine fault-tolerant algorithms through randomized testing. Despite its simplicity, ByzzFuzz is effective at discovering fault-tolerance bugs in the implementations of large-scale consensus systems. As far as we are aware, ByzzFuzz is the first automated testing algorithm that could discover previously unknown Byzantine fault tolerance bugs in the production implementations of blockchain systems.

ByzzFuzz provides a simple, practical, yet effective method for automated testing of the implementations of Byzantine fault-tolerant systems. It is applicable to any round-based distributed algorithm, most dominantly consensus algorithms, which are at the core of many distributed systems, including blockchains. We observe that the developers of fault-tolerant systems may ignore unexpected, faulty situations while implementing the algorithms. Therefore, the implementations of systems may not be fault tolerant despite the correctness of the protocol they implement. ByzzFuzz can help discover such bugs in simple test executions by injecting a *bounded* number of network and process faults into the *protocol rounds*, where process faults are modeled using *structure-aware* and *small-scope* mutations to protocol messages.

ByzzFuzz is an initial step in the randomized testing of Byzantine fault-tolerant systems. Future work will extend the randomized test generation to prioritize certain executions, apply sample set reduction techniques, or guide the test generation toward more complicated test case scenarios.

ACKNOWLEDGMENTS

We would like to thank Daniel Cason, Ivan Gavran, and Josef Widder for the insightful discussions about the Tendermint protocol. For their prompt and detailed feedback on the Ripple protocol executions and the bug fix, we thank Nik Bougalis, Elliot Lee, David Schwartz, and Lauren Weymouth. We also thank the anonymous reviewers for their insightful remarks and suggestions, which helped us improve the paper.

DATA AVAILABILITY STATEMENT

The implementation of ByzzFuzz and the source code of the systems under test are available in the accompanying artifact [Winter et al. 2023].

REFERENCES

Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *CoRR* abs/1712.01367 (2017). arXiv:1712.01367 http://arxiv.org/abs/1712.01367

Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016*, Marcos K. Aguilera, Brian Cooper, and Yanlei Diao (Eds.). ACM, 17–28. https://doi.org/10.1145/2987550.2987555

Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. https://doi.org/10.5555/3430915.3442427

Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 331–346. https://doi.org/10.1145/2723372.2723711

Benjamin Aminof, Sasha Rubin, Ilina Stoilkovska, Josef Widder, and Florian Zuleger. 2018. Parameterized Model Checking of Synchronous Distributed Algorithms by Abstraction. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10747)*, Isil Dillig and Jens Palsberg (Eds.). Springer, 1–24. https://doi.org/10.1007/978-3-319-73721-8_1

Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. 2020. Security Analysis of Ripple Consensus. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference) (LIPIcs, Vol. 184)*, Quentin Bramas, Rotem Oshman, and Paolo Romano (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:16. https://doi.org/10.4230/LIPIcs.OPODIS.2020.10

Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci Piergiovanni. 2019. Dissecting Tendermint. In *Networked Systems - 7th International Conference, NETYS 2019, Marrakech, Morocco, June 19-21, 2019, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11704)*, Mohamed Faouzi Atig and Alexander A. Schwarzmann (Eds.). Springer, 166–182. https://doi.org/10.1007/978-3-030-31277-0_11

Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolic, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 30:1–30:15. https://doi.org/10.1145/3190508.3190538

Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. 2015. Ripple: Overview and Outlook. In *Trust and Trustworthy Computing - 8th International Conference, TRUST 2015, Heraklion, Greece, August 24-26, 2015, Proceedings (Lecture Notes in Computer Science, Vol. 9229)*, Mauro Conti, Matthias Schunter, and Ioannis G. Askoxylakis (Eds.). Springer, 163–180. https://doi.org/10.1007/978-3-319-22846-4_10

Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. 2021. Twins: BFT Systems Made Robust. In *25th International Conference on Principles of Distributed Systems, OPODIS 2021, December 13-15, 2021, Strasbourg, France (LIPIcs, Vol. 217)*, Quentin Bramas, Vincent Gramoli, and Alessia Milani (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:29. https://doi.org/10.4230/LIPIcs.OPODIS.2021.7

Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. 2019. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep* 7 (2019).

Cory Bennett and Ariel Tseitlin. 2012. Chaos monkey released into the wild. *Netflix Tech Blog* 30 (2012), 1.

Christian Berger, Hans P. Reiser, and Alysson Bessani. 2021. Making Reads in BFT State Machine Replication Fast, Linearizable, and Live. In *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 1–12. https://doi.org/10.1109/SRDS53918.2021.00010

Nathalie Bertrand, Igor Konnov, Marijana Lazic, and Josef Widder. 2019. Verification of Randomized Consensus Algorithms Under Round-Rigid Adversaries. In *30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands (LIPIcs, Vol. 140)*, Wan J. Fokkink and Rob van Glabbeek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:15. https://doi.org/10.4230/LIPIcs.CONCUR.2019.33

Ethan Buchman. 2016. *Tendermint: Byzantine fault tolerance in the age of blockchains.* Ph. D. Dissertation. University of Guelph.

Ethan Buchman, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, Dragos-Adrian Seredinschi, and Josef Widder. 2022. Revisiting Tendermint: Design Tradeoffs, Accountability, and Practical Use. In *52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2022, Supplemental Volume, Baltimore, MD, USA, June 27-30, 2022*. IEEE, 11–14. https://doi.org/10.1109/DSN-S54099.2022.00014

Ethan Buchman, Jae Kwon, and Zarko Milosevic. 2018. The latest gossip on BFT consensus. *CoRR* abs/1807.04938 (2018). arXiv:1807.04938 http://arxiv.org/abs/1807.04938

Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010*, James C. Hoe and Vikram S. Adve (Eds.). ACM, 167–178. https://doi.org/10.1145/1736020.1736040

Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer. https://doi.org/10.1007/978-3-642-15260-3

Christian Cachin and Björn Tackmann. 2019. Asymmetric Distributed Trust. In *23rd International Conference on Principles of Distributed Systems, OPODIS 2019, December 17-19, 2019, Neuchâtel, Switzerland (LIPIcs, Vol. 153)*, Pascal Felber, Roy Friedman, Seth Gilbert, and Avery Miller (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:16. https://doi.org/10.4230/LIPIcs.OPODIS.2019.7

Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild. *CoRR* abs/1707.01873 (2017). arXiv:1707.01873 http://arxiv.org/abs/1707.01873

Christian Cachin and Luca Zanolini. 2021. Asymmetric Asynchronous Byzantine Consensus. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2021 International Workshops, DPM 2021 and CBT 2021, Darmstadt, Germany, October 8, 2021, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13140)*, Joaquín García-Alfaro, Jose Luis Muñoz-Tapia, Guillermo Navarro-Arribas, and Miguel Soriano (Eds.). Springer, 192–207. https://doi.org/10.1007/978-3-030-93944-1_13

Johnny Cao. 2020. A Practical Byzantine Fault Tolerance (PBFT) emulator built in Java. http://github.com/caojohnny/pbft-java.

Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, Margo I. Seltzer and Paul J. Leach (Eds.). USENIX Association, 173–186. https://dl.acm.org/citation.cfm?id=296824

Bernadette Charron-Bost and André Schiper. 2009. The Heard-Of model: computing in distributed systems with benign faults. *Distributed Comput.* 22, 1 (2009), 49–71. https://doi.org/10.1007/s00446-009-0084-6

Brad Chase and Ethan MacBrough. 2018. Analysis of the XRP Ledger Consensus Protocol. *CoRR* abs/1802.07242 (2018). arXiv:1802.07242 http://arxiv.org/abs/1802.07242

Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 536–547. https://doi.org/10.1145/3324884.3416548

Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 153–168. http://www.usenix.org/events/nsdi09/tech/full_papers/clement/clement.pdf

Domenico Cotroneo, Luigi De Simone, and Roberto Natella. 2022. ThorFI: a Novel Approach for Network Fault Injection as a Service. *J. Netw. Comput. Appl.* 201 (2022), 103334. https://doi.org/10.1016/j.jnca.2022.103334

Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. 2019. Communication-Closed Asynchronous Protocols. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11562)*, Isil Dillig and Serdar Tasiran (Eds.). Springer, 344–363. https://doi.org/10.1007/978-3-030-25543-5_20

Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. 2016. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016*, Angela Demke Brown and Florentina I. Popovici (Eds.). USENIX Association, 249–262. https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis

Alan J. Demers, Daniel H. Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard E. Sturgis, Daniel C. Swinehart, and Douglas B. Terry. 1987. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, Fred B. Schneider (Ed.). ACM, 1–12. https://doi.org/10.1145/41840.41841

Diem. 2021. Diem Core - network tests. https://github.com/diem/diem/blob/main/consensus/src/network_tests.rs.

Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. 2017. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 1085–1100. https://doi.org/10.1145/3035918.3064033

Cezara Dragoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. 2020. Testing consensus implementations using communication closure. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 210:1–210:29. https://doi.org/10.1145/3428278

Cezara Dragoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-Based Framework for Verifying Consensus Algorithms. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 161–181. https://doi.org/10.1007/978-3-642-54013-4_10

Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323. https://doi.org/10.1145/42282.42283

Tzilla Elrad and Nissim Francez. 1982. Decomposition of Distributed Programs into Communication-Closed Layers. *Sci. Comput. Program.* 2, 3 (1982), 155–173. https://doi.org/10.1016/0167-6423(83)90013-8

Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 2011. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, Thomas Ball and Mooly Sagiv (Eds.). ACM, 411–422. https://doi.org/10.1145/1926385.1926432

Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (1985), 374–382. https://doi.org/10.1145/3149.214121

Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. https://doi.org/10.1145/564585.564601

GitHub. 2020. Twins Simulator. https://github.com/asonnino/twins-simulator/tree/master/fhs.

GitHub. 2022. HotStuff. https://github.com/relab/hotstuff.

Patrice Godefroid. 1996. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. Lecture Notes in Computer Science, Vol. 1032. Springer. https://doi.org/10.1007/3-540-60761-7

Haryadi S Gunawi, Thanh Do, Joseph M Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. 2011a. Failure as a service (FAAS): A cloud service for large-scale, online failure drills. *University of California, Berkeley, Berkeley* 3 (2011).

Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011b. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

Divya Gupta, Lucas Perronne, and Sara Bouchenak. 2016. BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols. In *Distributed Applications and Interoperable Systems - 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9687)*, Márk Jelasity and Evangelia Kalyvianaki (Eds.). Springer, 115–128. https://doi.org/10.1007/978-3-319-39577-7_10

Apache Hadoop. 2009. Fault Injection Framework and Development Guide. https://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html.

Raluca Halalai, Thomas A. Henzinger, and Vasu Singh. 2011. Quantitative Evaluation of BFT Protocols. In *Eighth International Conference on Quantitative Evaluation of Systems, QEST 2011, Aachen, Germany, 5-8 September, 2011*. IEEE Computer Society, 255–264. https://doi.org/10.1109/QEST.2011.40

Yury Izrailevsky and Ariel Tseitlin. 2011. The Netflix simian army. *The Netflix Tech Blog* (2011).

Daniel Jackson and Craig Damon. 1996. Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector. *IEEE Trans. Software Eng.* 22, 7 (1996), 484–495. https://doi.org/10.1109/32.538605

Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. PREFAIL: a programmable tool for multiple-failure injection. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, Cristina Videira Lopes and Kathleen Fisher (Eds.). ACM, 171–188. https://doi.org/10.1145/2048066.2048082

Charles Edwin Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code (Awarded Best Paper). In *4th Symposium on Networked Systems Design and Implementation NSDI (2007), April 11-13, 2007, Cambridge, Massachusetts, USA, Proceedings*, Hari Balakrishnan and Peter Druschel (Eds.). USENIX. http://www.usenix.org/events/nsdi07/tech/killian.html

Minjeong Kim, Yujin Kwon, and Yongdae Kim. 2019. Is Stellar As Secure As You Think?. In *2019 IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 377–385. https://doi.org/10.1109/EuroSPW.2019.00048

Kyle Kingsbury. 2022. Jepsen. http://jepsen.io/.

Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 160:1–160:28. https://doi.org/10.1145/3276530

Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (1982), 382–401. https://doi.org/10.1145/357172.357176

Hyojeong Lee, Jeff Seibert, Md. Endadul Hoque, Charles Edwin Killian, and Cristina Nita-Rotaru. 2014. Turret: A Platform for Automated Attack Finding in Unmodified Distributed System Implementations. In *IEEE 34th International Conference on Distributed Computing Systems, ICDCS 2014, Madrid, Spain, June 30 - July 3, 2014*. IEEE Computer Society, 660–669. https://doi.org/10.1109/ICDCS.2014.73

Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank

Levy (Eds.). USENIX Association, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. 2021. Model-based testing of networked applications. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 529–539. https://doi.org/10.1145/3460319.3464798

Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 419–431. https://doi.org/10.1145/3173162.3177161

Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. 2019. Fast and secure global payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 80–96. https://doi.org/10.1145/3341301.3359636

Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. CrashTuner: detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 114–130. https://doi.org/10.1145/3341301.3359645

Rupak Majumdar and Filip Niksic. 2018. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.* 2, POPL (2018), 46:1–46:24. https://doi.org/10.1145/3158134

Toufik Mansour. 2013. *Combinatorics of set partitions*. CRC Press Boca Raton.

Rolando Martins, Rajeev Gandhi, Priya Narasimhan, Soila M. Pertet, António Casimiro, Diego Kreutz, and Paulo Veríssimo. 2013. Experiences with Fault-Injection in a Byzantine Fault-Tolerant Protocol. In *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8275)*, David M. Eyers and Karsten Schwan (Eds.). Springer, 41–61. https://doi.org/10.1007/978-3-642-45065-5_3

Lara Mauri, Stelvio Cimato, and Ernesto Damiani. 2020. A Formal Approach for the Analysis of the XRP Ledger Consensus Protocol. In *Proceedings of the 6th International Conference on Information Systems Security and Privacy, ICISSP 2020, Valletta, Malta, February 25-27, 2020*, Steven Furnell, Paolo Mori, Edgar R. Weippl, and Olivier Camp (Eds.). SCITEPRESS, 52–63. https://doi.org/10.5220/0008954200520063

David Mazieres. 2015. The Stellar Consensus Protocol: A federated model for internet-level consensus. *Stellar Development Foundation* 32 (2015).

Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, Carlo Curino, Georgia Koutrika, and Ravi Netravali (Eds.). ACM, 388–402. https://doi.org/10.1145/3472883.3487005

Atsuki Momose. 2019. Force-Locking Attack on Sync Hotstuff. *IACR Cryptol. ePrint Arch.* (2019), 1484. https://eprint.iacr.org/2019/1484

Yoram Moses and Sergio Rajsbaum. 2002. A Layered Analysis of Consensus. *SIAM J. Comput.* 31, 4 (2002), 989–1021. https://doi.org/10.1137/S0097539799364006

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 446–455. https://doi.org/10.1145/1250734.1250785

Srinidhi Nagendra. 2022. Netrix. https://netrixframework.github.io/.

Filip Niksic. 2019. *Combinatorial Constructions for Effective Testing*. Ph. D. Dissertation. Technische Universität Kaiserslautern.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 329–340. https://doi.org/10.1145/3293882.3330576

Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1629–1642. https://doi.org/10.1109/SP40000.2020.00067

Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 460–465. https://doi.org/10.1109/ICST46399.2020.00062

Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Trans. Software Eng.* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

Prashant Pogde, Siddharth Wagle, and Basavaraj M. 2020. Apache Ozone Fault Injection Framework. https://blog.cloudera.com/apache-ozone-fault-injection-framework/.

Casey Rosenthal. 2017. Principles of Chaos Engineering. USENIX Association, San Francisco, CA.

David Schwartz, Noah Youngs, Arthur Britto, et al. 2014. The Ripple Protocol Consensus Algorithm. *Ripple Labs Inc White Paper* 5, 8 (2014), 151.

Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT Protocols Under Fire. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, Jon Crowcroft and Michael Dahlin (Eds.). USENIX Association, 189–204. http://www.usenix.org/events/nsdi08/tech/full_papers/singh/singh.pdf

João Soares, Ricardo Fernandez, Miguel Silva, Tadeu Freitas, and Rolando Martins. 2021. ZERMIA - A Fault Injector Framework for Testing Byzantine Fault Tolerant Protocols. In *Network and System Security - 15th International Conference, NSS 2021, Tianjin, China, October 23, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13041)*, Min Yang, Chao Chen, and Yang Liu (Eds.). Springer, 38–60. https://doi.org/10.1007/978-3-030-92708-0_3

A. J. Stam. 1983. Generation of a Random Partition of a Finite Set by an Urn Model. *J. Comb. Theory, Ser. A* 35, 2 (1983), 231–240. https://doi.org/10.1016/0097-3165(83)90009-2

Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. 2022. Automatic Reliability Testing For Cluster Management Controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 143–159. https://www.usenix.org/conference/osdi22/presentation/sun

Tendermint. 2021. Tendermint: Tendermint Core (BFT Consensus) in Go (v0.34.7). https://github.com/tendermint/tendermint/tree/v0.34.7.

Tatsuhiro Tsuchiya and André Schiper. 2011. Verification of consensus algorithms using satisfiability solving. *Distributed Comput.* 23, 5-6 (2011), 341–358. https://doi.org/10.1007/s00446-010-0123-3

Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. 2023. Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2023, Melbourne, Australia, May 14-20, 2023*. IEEE. to appear.

Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. Randomized Testing of Byzantine Fault Tolerant Consensus Algorithms (artifact). https://zenodo.org/record/7510752. https://doi.org/10.5281/zenodo.7510752

XRPLF. 2021. *Decentralized cryptocurrency blockchain daemon implementing the XRP Ledger in C++ (v.1.7.2)*. https://github.com/XRPLF/rippled/tree/1.7.2.

Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 213–228. http://www.usenix.org/events/nsdi09/tech/full_papers/yang/yang.pdf

Qian Zhang, Jiyuan Wang, Muhammad Ali Gulzar, Rohan Padhye, and Miryung Kim. 2020. BigFuzz: Efficient Fuzz Testing for Data Analytics Using Framework Abstraction. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 722–733. https://doi.org/10.1145/3324884.3416641

Received 2022-10-28; accepted 2023-02-25