Hyperparameter-Tuned Randomized Testing for Byzantine Fault-Tolerance of the XRP Ledger Consensus Protocol

Aistė Macijauskaitė[1]

Supervisors: Dr. Burcu Kulahcioglu Ozkan[1]**, Dr. Mitchell Olsthoorn**[1]**, Dr. Annibale Panichella**[1]

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

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Aistė Macijauskaitė
Final project course: CSE3000 Research Project
Thesis committee: Dr. Burcu Kulahcioglu Ozkan, Dr. Mitchell Olsthoorn, Dr. Annibale Panichella, Dr. Jérémie Decouchant

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Hyperparameter-Tuned Randomized Testing for Byzantine Fault-Tolerance in the XRP Ledger Consensus Protocol

Aistė Macijauskaitė

Delft University of Technology
Delft, The Netherlands

Abstract

Blockchain systems rely on consensus protocols to ensure agreement among nodes even in the presence of malicious or faulty nodes. A consensus protocol that provides safety and liveness guarantees under such conditions is known as a *Byzantine fault-tolerant (BFT)* protocol. Various whitepapers describe the design and implementation of BFT protocols, providing formal proofs of their safety and liveness properties. However, in practice such protocols are difficult to implement correctly and often contain subtle logic errors that cause consensus violations. Ensuring correctness is especially important for the XRP Ledger—a public, decentralized blockchain that processes transactions for the widely used cryptocurrency XRP. Thorough testing of consensus protocols is crucial, but systematic testing is challenging and time-consuming due to the large number of possible network and timing configurations.

In this paper, we present a replication package for evaluating randomized Byzantine fault tolerance testing on the XRP Ledger Consensus Protocol (LCP). We implement the ByzzFuzz search algorithm and compare it to naive random testing. Additionally, we investigate the impact of different hyperparameter configurations on the performance of the ByzzFuzz algorithm. Our experimental results demonstrate that both naive random testing and the ByzzFuzz algorithm detect seeded bugs in the XRP LCP, with ByzzFuzz algorithm uncovering more agreement violations. Finally, we identify the most effective hyperparameter configurations for the ByzzFuzz algorithm.

1 Introduction

A core component of blockchain systems is the consensus protocol, which guarantees that all nodes in the underlying distributed system agree on a single, consistent transaction history, even in the presence of faulty or malicious processes. To be reliable, a consensus protocol must provide safety (i.e., no two nodes decide on conflicting outcomes) and liveness (i.e., the system continues to make progress) guarantees. These properties are particularly difficult to maintain in the presence of Byzantine faults, where nodes may deviate from the protocol and behave unpredictably or maliciously [1].

This paper presents a case study of Byzantine fault tolerance (BFT) testing for the XRP Ledger, a global enterprise payment network that processes millions of transactions per day across more than 40 countries. Given its critical role in the financial infrastructure, it is important to ensure that its consensus protocol is secure and resilient. Although XRPL Consensus Protocol (LCP) [2, 3] is designed to be Byzantine fault tolerant, its practical implementation is highly complex and has already been shown to contain bugs in its previous versions [4–6].

Prior work on the Byzantine fault tolerance search algorithm introduced a structured fault-injection strategy that systematically samples network and process faults while preserving protocol semantics [6]. This approach has proven effective at detecting Byzantine faults in distributed systems, including Tendermint and the XRP Ledger. Although ByzzFuzz search algorithm has shown promising results, there has been little focus on optimizing its hyperparameters.

In this paper, we aim to *investigate the impact of different hyperparameter configurations on the performance of the ByzzFuzz testing method.* Various parameters such as the number of rounds with network faults, the number of rounds with process faults, and the number of rounds in which faults are injected, can significantly affect test outcomes. For example, increasing the number of rounds with network faults can lead to more faults being introduced, but it can also reduce the chance of reaching valid protocol states. Conversely, a lower number of network faults might not explore the fault space sufficiently. Moreover, optimizing these parameters is often computationally expensive, as it requires many test executions to compare outcomes across different configurations. As demonstrated in the broader software testing literature, hyperparameter tuning can have a critical impact on algorithmic performance [7]. In contrast to prior work [6], which conducted only a small preliminary study, we perform a more comprehensive parameter configuration analysis to better understand how these values influence testing effectiveness.

To explore Byzantine fault tolerance bugs, we use two testing approaches that we have implemented within the Rocket testing framework [8]. First, we employ a naive random testing approach, which is a common technique for testing large-scale distributed systems due to its effectiveness in uncovering bugs in consensus protocols. The XRP LCP is tested by randomly introducing network faults, such as message drops to isolate processes or partition the network, and process faults, such as mutating message content to simulate malicious behavior. Second, we use a more systematic ByzzFuzz testing method [6], which performs targeted network and process fault injection. This testing method helps to navigate the space of possible process faults by introducing structure-preserving semantic message mutations. Such an approach is particularly effective at uncovering deep logic and implementation bugs that are less likely to be found through purely random testing. We evaluate both testing approaches on the current implementation (v2.4.0) of the XRP LCP and a modified version containing seeded bugs.

Our experimental results show that naive random testing is effective at detecting Byzantine fault tolerance bugs in large-scale blockchain systems. However, the ByzzFuzz testing algorithm, which

uses structurally-aware and semantically guided mutations, significantly improves fault detection performance. In particular, it detects a greater number of critical violations within fewer test executions, especially under certain optimized configurations.

This paper makes the following contributions:

- A case study evaluating the effectiveness of naive random testing and the ByzzFuzz algorithm for Byzantine fault tolerance on the XRPL blockchain system.
- A discussion of key challenges and opportunities for future research in search-based BFT testing.
- A replication package [9] that extends the Rocket testing framework with implemented ByzzFuzz testing method

2 Background and Related Work

Blockchains operate on a distributed ledger system, where transactions are recorded in a public ledger that is maintained by a network of computers (nodes). These transactions are grouped into blocks, which are linked together using cryptographic techniques to form an immutable chain ensuring both security and transparency in the system.

To maintain consistency across the network, nodes work together to validate and confirm transactions, group them into blocks, and add them to the blockchain in a consistent, totally ordered sequence. A set of rules describing how this coordination happens is called a consensus protocol. For the system to operate reliably, a consensus protocol must guarantee safety and liveness, even in the presence of network failures or malicious attacks. Such scenarios may involve attempting to disable the network from processing new transactions, committing fraudulent transactions, or launching double-spending attacks.

The robustness of consensus protocols becomes especially important when we consider the possibility of malicious actors, known as Byzantine nodes [1], that may behave arbitrarily or deliberately attempt to disrupt the network. Unlike honest nodes, Byzantine nodes may not respond to messages, send incorrect messages, and even send different messages to different parties. A consensus protocol that continues to function correctly despite the presence of such faulty nodes is called *Byzantine fault tolerant (BFT)*. However, implementing BFT protocols correctly in practice is notoriously difficult, as it is highly complex and even small mistakes can compromise its core safety and liveness guarantees.

2.1 XRPL Consensus Algorithm

XRP LCP differs from traditional BFT consensus protocols as it guarantees consistency with only partial agreement among nodes. This allows for faster consensus and lower transaction latency, making it suitable for high-performance applications like XRP Ledger. Instead of relying on full agreement from all nodes, XRP LCP uses an asymmetric trust model [10], where each process can choose which processes it trusts and which ones it considers faulty. This enables each validator node to independently define its own trusted subset of nodes, known as a *Unique Node List (UNL)*. During the consensus process, a validator only considers messages from the nodes in its own UNL.

The XRPL consists of a decentralized network of validator nodes, each maintaining a full copy of the ledger history, which is a record of all transactions that have occurred in the network.

During the consensus process, we first enter *Open Phase* where each validator collects new transactions from clients onto its *open ledger*. Validator nodes then broadcast `<TMTransaction, rawTransaction>` message through the entire network, where the "rawTransaction" field contains the hash of a transaction submitted by a client for validation.

After filling up the open ledger with transactions, we enter *Proposal Phase*. During this phase, each validator shares its candidate transaction set with its UNL by broadcasting `<TMProposeSet, proposeSeq, currentTxHash, previousLedger, nodePubKey, closeTime, signature>` message. Here "proposeSeq" represents the round number of the proposal, "currentTxHash" is the hash of the proposed transaction set to be committed in the next ledger, "previousLedger" is the hash of the last fully validated ledger, "nodePubKey" is the public key of the node used to sign the message, "closeTime" indicates the close time of the previous ledger and "signature" is used to verify the sender. During each sub-round, validators compare the proposals they receive from their UNL and adjust their own sets. Any transaction supported by at least a threshold of trusted validators remains in the set while other transactions are removed from consideration. This process is called *avalanche*, as the threshold increases with each round. Once the validator sees that at least 80% of its UNL agrees on the same transaction set, the consensus is said to be reached. If we fail to reach 80% of agreement within the predefined time limit, then an empty ledger is generated.

After the proposal round, the protocol enters the *Validation Phase*. In this phase, validators close the current ledger, which now contains the agreed-upon transaction set. Each validator sends `<TMValidation, validation>` to its UNL to determine whether the ledger can be finalized. The "validation" includes the hash of the closed ledger. Each node collects these validation messages and, once 80% of its trusted validators agree on the same hash, it considers the ledger fully validated. At this point, the ledger becomes final and its transactions are permanently committed. If we do not receive enough matching validation hashes, an empty ledger is created and the system proceeds to a new Open Phase to collect transactions for the next ledger.

This iterative process guarantees convergence [3], which is defined as the point where the network reaches a strong consensus on a ledger. That ledger is then accepted and becomes the last-closed ledger. Strong consensus refers to validators eventually agreeing on and committing new transactions to the ledger, rather than repeatedly generating empty ledgers. This is achieved while preserving the protocol's safety and liveness guarantees. The correctness of XRP LCP holds as long as the network consists of no more than $\lfloor (n-1)/5 \rfloor$ Byzantine nodes. It also requires the UNL-connectedness condition to be satisfied, which means that it needs at least 60% overlap between the UNLs of any two validator nodes [3].

2.2 Correctness Properties

Distributed consensus protocols are considered correct if they satisfy the following safety and liveness properties [11]:

(1) **Agreement**: Honest nodes decide identically.

(2) **Validity**: If an honest node decides a value, then that value was proposed by some other honest node.
(3) **Integrity**: No honest node in the network decides twice.
(4) **Termination**: Every honest node decides on a value eventually.

Safety violations occur when honest nodes make conflicting decisions. For example, if they agree on different versions of a ledger in the same consensus round, this would result in a fork in the network. Liveness violations could happen when the protocol fails to make progress, for instance, when a consensus round never terminates and nodes remain undecided, rendering the system unresponsive. A consensus protocol that satisfies all four properties is considered to guarantee both safety and liveness and therefore is considered to be Byzantine fault tolerant.

2.3 Related Work

There is a large body of work on systematically testing consensus protocols for distributed systems, including including dBug [12], MoDist [13], FlyMC [14]. However, these approaches suffer from state space explosion as they exhaustively explore all possible execution orders, and thus do not scale for large distributed systems. Alternatively, randomized testing methods have been proposed to improve scalability. Tools like Jepsen [15] and CoFI [16] test distributed systems by injecting random network faults to exercise system behavior under fault conditions. Other approaches introduce guided search algorithms such as probabilistic scheduling [17], and learning-based strategies such as evolutionary algorithms [18] to guide the test generation toward problematic executions. However, none of these methods specifically target Byzantine faults and thus cannot detect Byzantine fault-tolerance bugs.

Recent Byzantine fault tolerant testing tools for distributed systems have focused on the manual design of attack scenarios, as seen in Netrix [19] and ZERMIA [20]. Although these approaches are powerful and allow for control over test cases, they are highly time-consuming and require developers to construct system-specific scenarios. These approaches fail to scale to large distributed systems like XRP Ledger, where the number of possible Byzantine attack scenarios is too large for manual exploration.

Alternative to manual testing for Byzantine-fault tolerance bugs, Twins [21] introduced a systematic testing approach where multiple identical faulty processes or "twins" are injected into the system. These twins simulate Byzantine behavior by double voting, losing the internal process state, or sending conflicting messages to other nodes. However, the main limitation of Twins is that it exhaustively explores all possible combinations of when and how the twins could behave differently by sampling known faults. This requires iterating through every protocol round and varying communication patterns among nodes, which leads to state space explosion as the number of twins increases.

A more scalable approach is introduced by ByzzFuzz algorithm [6], which detects fault-tolerance bugs by randomly injecting network and process faults during executions. To help navigate a large space of possible network and process faults, ByzzFuzz algorithm introduces fault-based and round-based testing. This bounds the search space by selecting a subset of rounds in which to potentially inject faults, and then applying mutations or message drops within those rounds. Moreover, it reduces the large space of possible process faults by introducing structure-aware, small-scope mutations, which mutate messages while keeping them syntactically correct. These mutations include small semantic changes such as modifying a value (e.g. incrementing sequence number) or replaying old messages (e.g. old proposal message). ByzzFuzz algorithm has proven to be successful in practice, uncovering deep logic and implementation bugs in real-world protocols, which includes discovering a previously unknown implementation bug in the XRP Ledger Consensus Protocol (LCP) [6]. Thus ByzzFuzz algorithm effectiveness lies in its ability to systematically explore the space of possible network and process scenarios while providing guidance on how to mutate messages and which rounds to consider faulty.

3 Methodology

We aim to explore the Byzantine fault tolerance of the XRP Ledger Consensus Protocol (LCP) by injecting random faults into a simulated network. To achieve this, we consider two fault types: network faults, which model network partitions, and process faults, which model Byzantine node behavior (malicious or unpredictable messages). The number of possible execution scenarios is too large to test manually. Moreover, node communication is inherently nondeterministic, as the system's behavior depends on timing and network conditions, which are hard to predict or exhaustively explore. As a result, we need automated and randomized testing methods.

In this paper, we use two search algorithms: naive random search as a baseline testing approach and a more guided *ByzzFuzz* search algorithm. The next subsections describe how each algorithm was implemented and adapted for testing the XRP LCP.

3.1 Naive Random Search Algorithm

The naive random search algorithm serves as a baseline for our testing approach. It is usually recommended as a baseline testing algorithm because it is simple, fast, and surprisingly effective in practice.

In our implementation, we first randomly sample a set of Byzantine nodes such that the protocol would still theoretically satisfy the safety and liveness properties. Then we continually inject network and process faults during the execution of the XRP LCP. Each message event consists of a tuple <sender, receiver, message>, which represents a message from one node (the sender) to another node (the receiver). On each such event, we randomly decide whether to drop the message, mutate its content if the sender belongs to a Byzantine node-set, or let it pass through the network without any changes. Note that message mutation is done by randomly flipping a single bit in the payload.

However, the bit-flip will often result in a syntactically incorrect message, which means we will be exercising the protocol's parsing logic rather than its consensus logic. Nevertheless, this simple naive random search algorithm can quickly explore broad behaviors and catch some obvious bugs.

3.2 ByzzFuzz Search Algorithm

To explore more thoroughly and uncover potential bugs, we need to reduce search space and provide some guidance to the search

process. Thus we have implemented a guided ByzzFuzz search algorithm [6]. The ByzzFuzz algorithm was designed to test Byzantine fault tolerant protocols by injecting structured faults in a bounded way. It relies on four key concepts:

- **Fault-bounded testing:** We restrict the total number of faults (e.g. at most d network faults and c process faults). This was introduced based on the observation that even small disruptions in network timing or process behavior can lead to significant bugs in Byzantine fault-tolerant systems [6]. Thus focusing on a few faults that are likely to cause issues is often sufficient.
- **Round-based testing:** We structure faults in specific protocol rounds. That is, we select communication rounds in which all messages are either dropped or mutated. This allows us to deal with message retransmissions as the protocol is designed to retransmit messages in case it is not delivered to the receiver. Moreover, it ensures that we send incorrect messages to the same receiver in the same round, which makes it more likely to trigger forks in test executions.
- **Structure-aware mutations:** When corrupting a message, we ensure to keep it syntactically correct. Rather than flipping random bits, we apply semantic changes (e.g. altering transaction hash or sequence number). With such mutations, we can uncover deeper logic bugs in the protocol.
- **Small-scope mutations:** We keep mutations small by having slightly incorrect values near the correct ones. This way we increase the chances of falling within the expected range of valid values. For example, incrementing or decrementing a sequence number by one or replaying a recent valid value is more likely to be accepted and can expose subtle logic errors.

Together, all four of these key concepts help to navigate the large space of possible network and process faults.

ByzzFuzz Algorithm Implementation. In particular, the ByzzFuzz algorithm's implementation expects an input of three parameters:

- d, number of protocol rounds with network faults
- c, number of protocol rounds with process faults
- r, the number of protocol rounds among which the faults will be injected

Moreover, it uses a mutation set for possible mutations specific to the XRPL message types. The algorithm proceeds in two phases:

(1) *Initalization*
We begin by randomly sampling a set of Byzantine nodes, which is a subset of nodes that are allowed to behave maliciously. This subset is chosen such that the number of Byzantine nodes stays within the bounds necessary for safety and liveness, i.e., at most $\lfloor (n-1)/5 \rfloor$. Next, we sample d network and c process faults that will be injected during execution.
Network faults are modeled as `<round, partition>`. The "round" is the protocol round in which the network fault occurs. Moreover, partitions are modeled as a random split of the validator set into disjoint subsets. Here "partition" represents a subset of nodes that are not allowed to communicate with other sets for the duration of the round. If the sender and receiver nodes are not in the same "partition", the message is dropped.
Process faults are modeled as `<round, procs, seed>`. The "round" is the protocol round in which the process fault occurs, while "procs" is a subset of nodes that will receive the mutated message during the "round". If the sender is a Byzantine node and the receiver node belongs to "procs" subset, then we randomly choose a mutation using the mutation set provided for that message type. The "seed" ensures that we deterministically mutate the message content, so that the same mutation is applied to the same message in every execution.

(2) *Test Execution*
We intercept messages between sender and receiver nodes `<sender, receiver, message>` and check whether they match any entries in the fault set. If the message is sent in a round that has a network fault, we check if the sender and receiver nodes are in the same partition. If they are not, the message is dropped. Otherwise, if the round has a process fault, we check whether the sender is Byzantine and whether the receiver belongs to the set of processes that will receive the mutated message. If so, we analyze the message type and look up possible mutations for that type. If mutations are available, we randomly choose one from the set using the seed and apply it to the message. Finally, if the message is not dropped or mutated, it is passed through the network as is.

After a test run, we collect the execution trace and check XRPL's consensus properties (agreement, validity, integrity, termination) offline.

Process Fault Modelling for XRPL. We modeled process faults for XRPL using the structure-aware mutations in Table 1.

As explained in Section 2.1, validator nodes receive and broadcast transactions from the clients using message `TMTransaction`, then they enter proposal and validation rounds where they exchange `TMProposeSet` and `TMValidation` messages respectively. We consider two approaches to process faults: small-scope and any-scope mutations. The specific mutation types used in each approach are detailed in Section 4.1. In the any-scope approach, we apply mutations that deviate significantly from the original values but still remain syntactically valid. In contrast, the small-scope strategy mutates the message fields to a value that is close to the original value. Thus small-scope mutations help with boundary-testing, which can expose bugs that arbitrary changes (any-scope mutations) might miss.

4 Study Design

We test the XRP Ledger Consensus Protocol (LCP) for Byzantine fault–tolerance bugs. In this paper, we aim to answer the following research questions:

RQ1: Can the ByzzFuzz search algorithm detect bugs in the XRP protocol or its variants?
RQ2: How does the ByzzFuzz search algorithm compare to a baseline algorithm in bug detection?
RQ3: How does the selection of test parameters affect the performance of the ByzzFuzz search algorithm?

Table 1: Structure-Aware Mutations of XRPL Messages. Mutated values are primed and typeset in bold.

Original Message	Mutated Message
`<TMTransaction, rawTransaction, status>`	`<TMTransaction, rawTransaction′, status>`
`<TMProposeSet, proposeSeq, currentTxHash, nodePubKey, closeTime, signature, previousLedger>`	`<TMProposeSet, proposeSeq′, currentTxHash, ...>`
	`<TMProposeSet, proposeSeq, currentTxHash′, ...>`
`<TMValidation, validation>`	`<TMValidation, validation′>`

We evaluate the performance of the ByzzFuzz search algorithm on the current version (2.4.0) of the XRP LCP and its variant with seeded bugs. First, we run the baseline algorithm, which randomly injects network and process faults during the execution of the XRP LCP. Then we run the ByzzFuzz search algorithm with different parameters to compare its performance with the baseline algorithm. Each parameter configuration is executed using either small-scope or any-scope mutations.

4.1 Experimental Setup

We tested the XRP LCP v2.4.0 within Rocket testing framework [8], which simulates a local network of XRPL validator nodes in Docker containers and injects network or process faults during the execution of consensus protocol. The Rocket testing framework consists of two main components: the network interceptor and the controller. The network interceptor captures messages exchanged between the sender and receiver nodes and forwards them to the controller. Then the controller decides how to handle (drop, delay, or modify) the messages and returns them to the interceptor for delivery.

We have designed an XRPL network of seven nodes and defined three UNLs, each with 60% overlap to meet the UNL overlap requirements specified in the XRPL whitepaper [3]. In particular, nodes 1, 2, and 3 trusts UNL_1 = {1,2,3,4,5}, node 4 trusts UNL_2 = {2,3,4,5,6}, while nodes 5, 6 and 7 trust UNL_3 = {3,4,5,6,7}. In each run, one node is randomly selected as a Byzantine node, ensuring safety and liveness properties by satisfying the condition that at most $\lfloor (n-1)/5 \rfloor$ nodes are Byzantine.

We model process faults using structure-aware mutations described in Section 3.2, namely any-scope and small-scope mutation approaches implemented as follows:

- **Any-Scope Mutations**: Hash fields, namely `rawTransaction`, `currentTxHash`, `validation`, are replaced with a special *null byte hash* that represents an empty transaction set or "anti-ledger." Moreover, because the null byte hash is a well-formed hash value, the mutated message bypasses the XRP LCP algorithm for detecting Byzantine behavior and therefore does not raise any suspicion from receiver nodes. Sequence number fields (`proposeSeq`) are replaced with a uniformly random value from the set of $[1, 100]$.
- **Small-Scope Mutations**: Hash fields, namely `rawTransaction`, `currentTxHash`, `validation`, are replaced with random values from the set of previously logged transaction hashes respectively. Sequence number fields (`proposeSeq`) are either incremented or decremented with equal probability.

Immediately after the network startup, we perform Genesis transactions to set up the accounts. Each account is initialized with a balance of 100,000 XRP.

During test execution, we submit four concurrent transactions to the XRPL network:

- **Tx1** = $\langle 1, 1, 2, 80\,000 \rangle$
- **Tx2** = $\langle 2, 1, 3, 81\,000 \rangle$
- **Tx3** = $\langle 3, 1, 3, 82\,000 \rangle$
- **Tx4** = $\langle 4, 1, 2, 83\,000 \rangle$

Here each transaction is represented as a tuple, which is ⟨ID, sender, receiver, amount⟩. Account 1 attempts to spend its funds twice on Account 2 and twice on Account 3 but its balance allows only one successful transfer. Transactions are submitted to different XRPL nodes simultaneously 2 seconds after the test starts. Moreover, we submit Tx1 to Node 1, Tx2 to Node 2, Tx3 to Node 6, and Tx4 to Node 7. It is important to note that Nodes 1 and 2 belong to UNL_1 and Nodes 6 and 7 to UNL_3. This setup allows us to evaluate whether the XRP LCP can handle concurrent transactions and whether it can prevent double-spending attacks.

After submitting transactions to the network, we wait for these transactions to be validated which under normal circumstances happens within the first 8 iterations. Then we run a test case for an additional 6 iterations to allow the network to heal.

We tested two versions of XRP LCP:

- **XRP LCP v2.4.0**: unmodified version of XRPL source code [22], which is a production-ready implementation of XRP LCP.
- **XRP LCP v2.4.0 with a seeded bug**: a modified version of the XRPL source code [23], where the threshold to validate proposals is set to 40% agreement instead of 80% agreement.

After running each test case, we collect the execution trace and check four consensus properties:

- **Agreement**: An execution violates the agreement if any two honest nodes validate different ledgers for the same round. At the end of each run, we check each node's validated ledger hashes that are stored in memory. If there is a mismatch between the ledger hashes for the same round, we record a violation.
- **Validity**: An execution violates validity if an honest node decides on a value that no other honest node proposed. We log all `TMTransaction` messages during execution and, at the end of each run, verify that every transaction hash in an honest node's memory matches one of the transactions originally proposed by an honest node.
- **Integrity**: An execution violates integrity if an honest node decides on the same value twice. We check each honest node's sequence of validated ledger hashes to ensure that the sequence numbers increase by exactly one at each step.
- **Termination**: An execution violates termination if an honest node does not decide on a value within a predetermined

time-bound. We set this upper bound at 90 seconds since under normal conditions the XRPL LCP should reach consensus within 60 seconds. An execution is considered correct only if all honest nodes decide within this time. Thus if any honest node stalls or misses the deadline, we record a termination violation.

For each XRPL LCP version, we run our baseline algorithm, which is a random search algorithm, and ByzzFuzz search algorithm using both small-scope and any-scope mutation approaches. For the random search algorithm, we vary the probabilities of dropping and mutating messages as described in section 3.1. For ByzzFuzz search algorithm, we adjust the parameters d, c, r, as described in section 3.2. These parameters control the number of protocol rounds with network and process faults, as well as the range of rounds in which faults are injected.

We run each search algorithm 100 times for every parameter configuration and record whether one or more consensus properties were violated. Tests are executed multiple times in order to account for the randomness of search algorithms as suggested by the existing guidelines [24]. Then we compare the algorithms by their success rate, in particular, by the number of runs in which they detect a violation or successfully uncover a bug. Also, it is important to note that too many network or process faults would cause nodes to stop responding, which in turn would always trigger termination violations. Thus, we limit the number of fault injections in order not to be biased toward results with termination violations.

5 Results

We evaluated the effectiveness of the baseline random testing method and the ByzzFuzz search algorithm in a seven-node network, which includes one Byzantine node. Each experiment ran for 100 test iterations over 14 protocol rounds, with faults injected during the first 8 rounds, allowing the network to heal during the remaining 6 rounds. We tested two fault-injection strategies: baseline random testing strategy (10% message drop and 10% message mutation) and the ByzzFuzz search algorithm, configured with varying parameters: $d = [0, 2]$ rounds with network faults and $c = [0, 2]$ rounds with process faults, distributed across the $r = 8$ fault-injection rounds. Both mutation approaches were used: small-scope (ss) and any-scope (as).

Table 2 presents the outcomes for the unmodified XRP LCP v2.4.0 and Table 3 shows outcomes for a version with a known seeded bug. Each table cell reports the number of runs in which a consensus property was violated: termination (T), validity (V), integrity (I), or agreement (A). Moreover, the last column reports the total number of runs with at least one violation. Note that a single run can yield multiple violations. For example, a run can violate both agreement and termination properties if the processes diverge in their decisions and later become stuck without progressing further.

Effectiveness of naive random testing for detecting bugs in XRP LCP. In the unmodified XRP LCP (Table 2), termination violations occurred in 65 runs, with no violations of validity, integrity, or agreement. In the seeded version of the protocol (Table 3), termination failed in 35 runs, and the seeded bug caused an agreement violation in just 3 runs. Interestingly, the number of termination violations is significantly higher in the unmodified version of the XRP LCP (65/100) compared to the seeded version (35/100).

This pattern is consistent with prior work [6], which has shown that naive random testing methods can lead to a large number of termination violations because random faults often crash processes early and therefore lead to premature protocol termination. Thus baseline random testing is not effective at detecting subtle logic errors in the protocol, as it tends to halt execution before allowing us to explore the deeper states of execution scenarios.

In summary, the baseline random testing method showed that the XRP LCP implementation frequently failed to make progress under random faults and almost never violated safety properties.

Effectiveness of the ByzzFuzz search algorithm for detecting bugs in XRP LCP. In contrast, the ByzzFuzz search algorithm detected more safety violations and generally fewer termination violations, although the overall count of termination violations remained substantial. On the seeded version of the XRP LCP (Table 3), the ByzzFuzz search algorithm was able to detect the seeded bug more efficiently than baseline testing (i.e., it required fewer executions to trigger a violation), and it did so in nearly all parameter configurations. In particular, for the parameter configuration $c = 1, d = 2$, the ByzzFuzz algorithm produced 10 agreement violations with small-scope mutations and 14 with any-scope, which is substantially more compared to only 3 runs total by the baseline. Moreover, the ByzzFuzz search algorithm tended to generate fewer termination violations than the baseline, indicating that the protocol could run longer and explore more execution scenarios before terminating.

We further compared the effectiveness of the small-scope and any-scope mutation strategies within the ByzzFuzz search algorithm. However, we did not observe a clear trend in the number of violations detected by either approach. For example, in Table 3, under the $(c = 1, d = 2)$ fault configuration, any-scope mutations detected 14 agreement violations, while small-scope mutations found 10 agreement violations. In contrast, under the $(c = 1, d = 1)$ fault configuration, any-scope mutations detected 3 agreement violations, while small-scope mutations found 6 agreement violations. This variability in results can be attributed to the inherent concurrency nondeterminism of the protocol. Thus we cannot conclude which approach is more effective, as it depends on the specific test case and the order of message delivery.

We found that the configuration $c = 1$, $d = 2$ produced the highest combined count of termination and agreement violations in both protocol versions and across both mutation scopes (small-scope and any-scope mutations). Moreover, the configuration $c = 0$, $d = 2$ also detected a high number of agreement violations but found less or the same number of termination violations in both protocol versions and across both mutation scopes. Although distributed consensus testing is inherently nondeterministic, these results consistently suggest that the $(1, 2)$ and $(0, 2)$ configurations are the most effective for evaluating XRP LCP, as they detect the highest number of violations in the fewest executions.

In conclusion, experimental results support the effectiveness of the ByzzFuzz search algorithm for both any-scope and small-scope mutation approaches as it allows deeper exploration of execution scenarios and uncovers more subtle logic bugs compared to the baseline algorithm.

Table 2: Testing XRP LCP v2.4.0 using small-scope (ss) and any-scope (as) mutations with varying rounds of c process faults and d network partitions.

Faults	T		V		I		A		Total	
baseline	65		0		0		0		65	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 0, d = 1$	41	31	0	0	0	0	0	0	44	31
$c = 0, d = 2$	37	44	0	0	0	0	0	0	37	44
$c = 1, d = 0$	31	29	0	0	0	0	0	0	31	29
$c = 1, d = 1$	36	36	0	0	0	0	0	0	36	36
$c = 1, d = 2$	40	44	0	0	0	0	0	0	43	44
$c = 2, d = 0$	36	39	0	0	0	0	0	0	36	39
$c = 2, d = 1$	45	34	0	0	0	0	0	0	45	34
$c = 2, d = 2$	54	42	0	0	0	0	0	0	56	42

Table 3: Testing seeded version of XRP LCP v2.4.0 using small-scope (ss) and any-scope (as) mutations with varying rounds of c process faults and d network partitions.

Faults	T		V		I		A		Total	
baseline	35		0		0		3		35	
	ss	as	ss	as	ss	as	ss	as	ss	as
$c = 0, d = 1$	0	1	0	0	0	0	4	6	4	7
$c = 0, d = 2$	2	2	0	0	0	0	13	8	15	8
$c = 1, d = 0$	2	1	0	0	0	0	4	1	4	2
$c = 1, d = 1$	0	0	0	0	0	0	6	3	6	3
$c = 1, d = 2$	4	5	0	0	0	0	10	14	13	17
$c = 2, d = 0$	0	0	0	0	0	0	2	1	2	1
$c = 2, d = 1$	2	0	0	0	0	0	3	7	4	7
$c = 2, d = 2$	3	3	0	0	0	0	7	8	9	10

6 Discussion

Our experiments have detected several consensus violations in the XRP Ledger Consensus Protocol (XRP LCP). In particular, we detected termination violations in both the production and seeded versions of the protocol, as well as agreement violations in the seeded version. These findings confirm the effectiveness of our search algorithms, which successfully uncovered that the proposal threshold was incorrectly set to 40% instead of the default 80%. The violations are closely linked to the structure of the network.

The network consisted of seven nodes organized into three overlapping Unique Node Lists (UNLs) designed to satisfy the 60% overlap requirement. The trust relationships were configured as follows: nodes 1, 2, and 3 trusted UNL_1 = 0, 1, 2, 3, 4, node 4 trusted UNL_2 = 1, 2, 3, 4, 5, and nodes 5, 6, and 7 trusted UNL_3 = 2, 3, 4, 5, 6.

Termination violations. Termination violations were detected in both the production and seeded versions of the protocol. These violations typically occurred when communication problems caused honest nodes to remove peers from their UNLs. For example, if too many messages between node A and node B were dropped or mutated in a way that caused suspicion, node A might remove node B from its UNL and stop sending messages. However, node B would still expect messages from node A and thus would lose a vote during the proposal and validation phases.

Given that each UNL contains five members and the quorum threshold is 80%, each node requires four out of five trusted nodes to agree in order to make progress. If two nodes stop communicating with a third, that third node cannot reach quorum and becomes stuck.

As a result, the network often splits into two clusters. For example, the cluster of nodes using UNL_1 (nodes 1–3) might stop progressing while the UNL_3 cluster (nodes 5–7) continued to reach consensus, or vice versa. In some scenarios, a single node became completely isolated and was unable to make progress. When the quorum threshold was reduced to 40% in the seeded version, termination violations occurred less frequently. The lower threshold meant that only two of the five trusted nodes were needed to form a quorum, so nodes could still reach consensus despite losing communication with other nodes.

Agreement violations. Agreement violations were observed only in the seeded version of the protocol. In these cases, different clusters of honest nodes finalized different ledger values. For instance, nodes in UNL_1 might agree on one ledger, while nodes in UNL3 agreed on a different one. This divergence included several cases:

one group validated a ledger with transactions while the other validated an empty ledger, both groups validated ledgers with different transactions, or both validated empty ledgers but with different ledger hashes, indicating a network split.

Moreover, node 4, which trusts UNL_2 and therefore overlaps with both UNL_1 and UNL_3, would end up with either set of nodes depending on how the consensus evolved.

These agreement violations were a direct result of the reduced proposal threshold. With a 40% quorum requirement (two of five), small isolated subgroups could independently achieve quorum and make decisions. For example, nodes 1, 2, and 3 along with node 4 could form a quorum under UNL_1 and commit to one ledger, while nodes 5, 6, and 7 could form a separate quorum under UNL_3 and commit to a conflicting ledger.

ByzzFuzz search algorithm suggested improvement. The original ByzzFuzz search algorithm pseudocode [6] first checks for network faults (message drops) before process faults (message corruptions). However, this means dropping happens more often than mutating because once a message qualifies for being dropped, we no longer check if it should be mutated. In a case where the sender node is Byzantine, and both a network fault and a process fault are applicable in the same round, the message always gets dropped and not mutated. To eliminate bias towards dropping messages, we introduce a new data structure `orderFirstNet`, which records per each round whether to check network or process faults first. Each round's priority is determined by a fair 50/50 coin flip.

These changes ensure reproducibility as each round independently has a 50% chance of prioritizing network faults and a 50% chance of prioritizing process faults. When a round prioritizes network faults, any qualifying message is dropped. Thus we consistently drop messages between isolated partitions in that round. When a round prioritizes process faults, we mutate all qualifying messages, causing the Byzantine node to apply the same corruption to the same recipients.

With these changes, the algorithm makes a reproducible decision between checking network faults first or processing faults first. Thus it remains fully reproducible under the same seed. The original pseudocode and the improved version incorporating these changes are provided in Appendix A as Algorithm 1 and Algorithm 2.

7 Threats to Validity

Internal validity. The main threat to internal validity is the nondeterminism inherent in randomized testing methods. To address this threat, we ran each randomized algorithm multiple times with every parameter configuration and based our conclusions on overall performance across all runs. Another potential threat is the correctness evaluation of test cases, specifically validity check. In particular, we were unable to extract information about proposed transactions from the proposal messages exchanged between nodes. Thus, to mitigate this, we have approximated the missing information by checking transaction hashes in transaction messages. However, in some cases, this approximation might be insufficient to determine whether a transaction was actually proposed.

External validity. The naive random testing algorithm and the ByzzFuzz search algorithm were specifically designed for testing the XRP LCP. Therefore, the results of our case study may not generalize to other consensus protocols. Further research is needed to adapt these algorithms to other consensus protocols and evaluate their performance.

8 Conclusion and Future Work

In this paper, we extended the Rocket testing framework by implementing the ByzzFuzz search algorithm, which guides fault injection more strategically. We also presented a case study on the effectiveness of the ByzzFuzz algorithm for Byzantine fault tolerance on the XRP Ledger Consensus Protocol compared to naive random testing. Our results show that both naive random testing and the ByzzFuzz algorithm can uncover bugs in the protocol, particularly agreement violations within a seeded XRP LCP version. In our evaluation, we demonstrated that the ByzzFuzz search algorithm significantly improves testing efficiency, detecting a higher number of critical violations per time unit compared to the baseline method. Additionally, we evaluated a broad range of hyperparameter configurations for the ByzzFuzz search algorithm.

Our findings consistently suggest that configurations $c = 1, d = 2$ and $c = 0, d = 2$ are the most effective for testing the XRP LCP. Here, c represents the number of process fault rounds, and d the number of network fault rounds. These configurations detect the highest count of termination and agreement violations in the fewest executions, across both mutation scopes and protocol versions. Although this study centers on the XRP LCP, the testing techniques and findings provide a foundation that can potentially be applied to other consensus protocols. Future work should include further experiments to determine which mutation scope, small-scope or any-scope, yields more efficient testing. Furthermore, an even wider range of hyperparameter configurations could be explored. Additionally, the performance of the improved ByzzFuzz search algorithm could be compared against previous version to determine whether it can improve testing efficiency.

9 Responsible Research

To ensure reproducibility, we have made the code for the ByzzFuzz search algorithm available on GitHub. We also provide a Docker image containing the testing framework, which allows to run the tests in a controlled environment. Detailed descriptions of the experimental setup are documented both in this paper and in the public repository. This includes parameter configurations, test execution details, detected violations, and random seeds per each test case, which allows to reproduce the results of each of the experiments.

An important ethical concern when testing consensus protocols is the potential impact on real funds and network stability. Thus, all experiments were conducted on a simulated local network of XRPL validator nodes and no production XRP Ledger network was used. Our goal was to improve the protocol's correctness and robustness, not to exploit vulnerabilities. For any discoveries of serious issues, we follow principles of Coordinated Vulnerability Disclosure (CVD). If potential bug is discovered, we will report it to the XRPL developers and give them sufficient time to fix the issues before disclosing them publicly. This allows critical systems to be updated before any findings from our experiments could be exploited.

References

[1] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[2] Bradley Chase and Ethan MacBrough. Analysis of the xrp ledger consensus protocol. *CoRR*, abs/1802.07242, 2018.

[3] David Schwartz, Noah Youngs, Arthur Britto, et al. The ripple protocol consensus algorithm. White Paper 8, Ripple Labs Inc., 2014.

[4] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? In *Proc. ACM Program. Lang.*, volume 2, pages 46:1–46:24, 2018.

[5] Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. Security analysis of ripple consensus. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, volume 184 of *LIPIcs*, pages 10:1–10:16, 2020.

[6] Levin N. Winter, Florena Busè, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proceedings of the ACM on Programming Languages (PACMPL)*, 7(OOPSLA(1)):757–788, 2023.

[7] Andrea Arcuri and Xin Yao. A theoretical and empirical study of the role of parameter settings for the (1+λ) evolutionary algorithm. *Information Sciences*, 181(6):1462–1492, 2011.

[8] W. Kanhai, I. van Loon, Y. Mangalgi, T. van der Valk, L. Witte, A. Panichella, M. Olsthoorn, and B. Kulahcioglu Ozkan. diseb-lab/rocket: v1.0.0, 2025. Software. 10.5281/zenodo.14865112.

[9] Calin Ciocănea, Atour Mousavi Gourabi, Wishaal Kanhai, Aistė Macijauskaitė, and Bryan Wassenaar. Rocket with Evolutionary and Byzzfuzz Testing Methods. https://github.com/amousavigourabi/rocket/tree/byzzfuzz, 2025. Accessed: 2025-06-16.

[10] Christian Cachin and Bryan Tackmann. Asymmetric distributed trust. In Philippe Felber, Robbert Friedman, Sébastien Gilbert, and Alexander Miller, editors, *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*, volume 153 of *LIPIcs*, pages 7:1–7:16, Neuchâtel, Switzerland, December 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[11] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, 2 edition, 2011.

[12] Jiri Simsa, Randal Bryant, and Garth A. Gibson. dbug: Systematic evaluation of distributed systems. In Ralf Huuck, Gerwin Klein, and Bernhard Schlich, editors, *5th International Workshop on Systems Software Verification (SSV '10)*, Vancouver, BC, Canada, October 2010. USENIX Association.

[13] Jie Yang, Tongping Chen, Min Wu, Zhen Xu, Xiaoqiao Liu, Hongyu Lin, Mingyang Yang, Fenglong Long, Li Zhang, and Lidong Zhou. Modist: transparent model checking of unmodified distributed systems. In Jennifer Rexford and Emin Gün Sirer, editors, *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2009)*, pages 213–228, Boston, MA, USA, April 2009. USENIX Association.

[14] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, David Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In George Candea, Robbert van Renesse, and Christian Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 20:1–20:16, Dresden, Germany, March 2019. ACM.

[15] Kathryn Kingsbury. Jepsen. http://jepsen.io/, 2016. [Online; accessed 2025-06-19].

[16] Haopeng Chen, Wei Dou, Dawei Wang, and Feng Qin. Cofi: Consistency-guided fault injection for cloud systems. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*, pages 536–547, Melbourne, Australia, September 2020. IEEE.

[17] Sidhartha Burckhardt, Prashant Kothari, Madanlal Musuvathi, and Sriram Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In Josep Torrellas and Vivek Sarkar, editors, *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2010)*, pages 167–178, Pittsburgh, Pennsylvania, USA, March 2010. ACM.

[18] Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. Evolutionary approach for concurrency testing of ripple blockchain consensus algorithm. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE SEIP 2023)*, Melbourne, Australia, May 2023. IEEE. to appear.

[19] Srinidhi Nagendra. Netrix. https://netrixframework.github.io/, 2022. Accessed: 2025-06-04.

[20] João Soares, Ricardo Fernandez, Miguel Silva, Tadeu Freitas, and Rolando Martins. ZERMIA: A fault injector framework for testing byzantine fault tolerant protocols. In Min Yang, Chao Chen, and Yang Liu, editors, *Network and System Security – 15th International Conference, NSS 2021, Tianjin, China, October 23, 2021, Proceedings*, volume 13041 of *Lecture Notes in Computer Science*, pages 38–60. Springer, 2021.

[21] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: BFT systems made robust. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:29. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2021.

[22] XRPLF. Decentralized cryptocurrency blockchain daemon implementing the xrp ledger in c++ (v2.4.0). Available at: https://github.com/XRPLF/rippled/tree/2.4.0, 2025. Accessed: 2025-05-25.

[23] Calin Ciocănea, Atour Mousavi Gourabi, Wishaal Kanhai, Aistė Macijauskaitė, and Bryan Wassenaar. Docker image: docker-rippled/seeded-2.4.0-fully-lowered-threshold. Available at: https://github.com/amousavigourabi/docker-rippled/pkgs/container/docker-rippled%2Fseeded-2.4.0-fully-lowered-threshold, 2025. Accessed: 2025-06-01.

[24] Andrea Arcuri and Lionel C. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

A Pseudocodes of Fault-Injection Algorithms

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

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
Data:
`networkFaults` : Set$\big[$(Int, Set[$\mathcal{P}$])$\big]$ ▷ round and partition of $\mathcal{P}$
`procFaults` : Set$\big[$(Int, Set[$\mathcal{P}$], Int)$\big]$ ▷ round, subset of $\mathcal{P}$, and seed

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

Algorithm 2 Random injection of c process faults and d network partition faults over r protocol rounds. Each round is assigned to either prioritize network or process faults with equal probability.

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
Data:
`networkFaults` : Set[(Int, Set[$\mathcal{P}$])] ▷ round and partition of $\mathcal{P}$
`procFaults` : Set[(Int, Set[$\mathcal{P}$], Int)] ▷ round, subset of $\mathcal{P}$, and seed
`orderFirstNet`: Set[Int]

1: **procedure** onInit
2: /* sample network faults */
3: `networkFaults` $\leftarrow \emptyset$
4: **for** $i \leftarrow 1$ to d **do**
5: round $\leftarrow$ `randomElementFrom`$([1, r])$
6: partition $\leftarrow$ `randomPartitionOf`$(\mathcal{P})$
7: `networkFaults` += {(round, partition)}
8: **end for**
9: /* sample per-round ordering */
10: `orderFirstNet` $\leftarrow$ { t | $t \in [1..r]$, `randomBool`() }
11: /* sample process faults */
12: $p_{byz} \leftarrow$ `randomElementFrom`$(\mathcal{P})$
13: `procFaults` $\leftarrow \emptyset$
14: **for** $i \leftarrow 1$ to c **do**
15: round $\leftarrow$ `randomElementFrom`$([1, r])$
16: procs $\leftarrow$ `randomSubsetOf`$(\mathcal{P})$
17: seed $\leftarrow$ `randomElementFrom`(Z)
18: `procFaults` += {(round, procs, seed)}
19: **end for**
20: **end procedure**
21: **procedure** onMessage(m)
22: **if** t = `roundOf`$(m) \in$ `orderFirstNet` **then**
23: **if** (`rnd`$(m), \pi) \in$ `networkFaults` **and** `isolates`$(\pi,$ `sender`$(m),$ `recv`$(m))$ **then**
24: /* do nothing, drop the message */
25: **else if** `sender`$(m) = p_{byz}$ **and** ((`rnd`$(m),$ `recv`$(m) \cup \_,$ seed) $\in$ `procFaults`) **then**
26: /* mutate and send the message */
27: $M \leftarrow$ `mutate`$(m,$ seed$)$
28: send(recv(m), M)
29: **else**
30: send(recv(m), m)
31: **end if**
32: **else**
33: **if** `sender`$(m) = p_{byz}$ **and** ((`rnd`$(m),$ `recv`$(m) \cup \_,$ seed) $\in$ `procFaults`) **then**
34: /* mutate and send the message */
35: $M \leftarrow$ `mutate`$(m,$ seed$)$
36: send(recv(m), M)
37: **else if** (`rnd`$(m), \pi) \in$ `networkFaults` **and** `isolates`$(\pi,$ `sender`$(m),$ `recv`$(m))$ **then**
38: /* do nothing, drop the message */
39: **else**
40: send(recv(m), m)
41: **end if**
42: **end if**
43: **end procedure**