EvoPriority

Evaluating Fitness Functions in Priority-Based Evolutionary Testing for the XRP Ledger Consensus Protocol

Călin Ciocănea

Supervisor(s): Burcu Kulahcioglu Ozkan, Annibale Panichella, Mitchell Olsthoorn

[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 8, 2023

Name of the student: *Călin Ciocănea*
Final project course: CSE3000 Research Project
Thesis committee: *Burcu Kulahcioglu Ozkan, Annibale Panichella, Mitchell Olsthoorn, Jérémie Decouchant*

EvoPriority: Evaluating Fitness Functions in Priority-Based Evolutionary Testing for the XRP Ledger Consensus Protocol

Călin Ciocănea

Technical University of Delft
Delft, Netherlands

ABSTRACT

The XRP Ledger Consensus Protocol is a Byzantine fault-tolerant algorithm that enables the XRP Ledger to reach agreement on which transactions to apply, supporting millions of transactions daily. While the protocol is correct by design, its practical implementation is vulnerable to concurrency-related bugs triggered by nondeterministic message delivery between distributed validator nodes. These bugs are subtle and difficult to expose through conventional testing. In this paper, we investigate the use of evolutionary concurrency testing combined with a priority-based message scheduling strategy to explore different message interleavings. Specifically, we evaluate multiple fitness functions and assess their ability to guide the search toward buggy executions. Our results show that EvoPriority, which applies evolutionary testing to priority-based schedules, is difficult to guide toward buggy executions regardless of the fitness function used. Although it is capable of uncovering violations on a bug-seeded versions of the XRP Ledger Consensus Protocol, its performance is similar to randomized concurrency testing.

1 INTRODUCTION

The XRP Ledger (XRPL) is a decentralized public blockchain that powers the XRP cryptocurrency. Over 50 billion XRP tokens are in circulation today, equating to more than 120 billion USD, with around 1.3 billion dollars worth being traded every day [6]. With such a high volume, one small flaw in the system could have massive implications. This is why the silent threat of implementation level bugs grows louder in the context of high-volume decentralized networks.

With over 300 institutional partners across more than 40 countries [10], the XRPL is one of the most widely adopted enterprise blockchain platforms. It functions as a decentralized network of computers, called nodes, which must maintain a common view of the ledger. In practice, this means nodes must agree on the set, order and content of transactions applied to the ledger, as well as the accounts involved in these transactions. At the core of this process lies the XRP Ledger Consensus Protocol (XRPL CP), a Byzantine fault-tolerant protocol that helps the XRPL achieve agreement across the network. It defines specific rules that allow nodes to exchange messages and converge on a common ledger state.

Unlike traditional blockchains such as Bitcoin, it does not rely on proof-of-work. Instead, XRPL CP uses a synchronized execution among nodes through a tightly timed sequence of consensus rounds. This design enables high throughput and low latency and, under ideal conditions, has been proven correct by design [16]. However, this theoretical robustness assumes stable network conditions and predictable message timing. In practice, even small variations in message timing or ordering, combined with possibly faulty or malicious nodes, can trigger concurrency issues. These issues could lead to consensus violations such as forks in the network, where different subsets of nodes agree on different versions of the ledger. Detecting such violations requires testing the XRPL CP under a range of message orderings and analysing how some orderings might trigger consensus violations.

Prior work by Van Meerten et al. [18] demonstrated the effectiveness of evolutionary algorithms for uncovering concurrency bugs in the XRPL CP. Their approach tested different fitness functions using a delay-based event representation, where the timing of message deliveries is manipulated. While their results show that evolutionary testing outperforms the random delay strategy, they did not investigate how these fitness functions behave when used with a priority-based strategy, where messages are reordered based on assigned priorities rather than delayed. The question remains: which fitness functions best guide the evolutionary search when applied to priority-based testing?

In this paper, we address this gap by applying evolutionary algorithms to priority-based testing of the XRPL CP. We refer to this approach as **EvoPriority**. Priority-based testing assigns different priorities to messages and then delivers each message in order of their priority, allowing exploration of different message interleavings. We use the EvoPriority approach to guide the bug search process by assigning scores to individual test cases, allowing the algorithm to generate new test cases based on the highest scoring ones. We rank test cases using different fitness functions in order to evaluate which one guides the evolutionary algorithm to the most bugs found over all generations. We evaluate two fitness functions, each emphasizing different aspects of XRPL CP behaviour: (1) TimeFitness, which favours test cases with longer execution times; (2) ProposalFitness, which rewards test cases requiring more proposal messages to reach consensus. To evaluate the effectiveness of each fitness function, we compare them to each other and to a baseline strategy, which we refer to as RandomPriority. This baseline randomly assigns priorities to messages, producing interleavings without evolutionary guidance. We evaluate the performance of each execution by the number of violations found over a fixed number of generations. All experiments are conducted within the Rocket framework [15], a system-level fuzzing platform that allows fine-grained control over message delivery in XRPL test networks.

In our evaluation, we applied priority-based testing to the XRPL CP using both RandomPriorty and EvoPriority strategies. While all approaches were able to detect a small number of violations in the bug-seeded version, none found any issues in the original (currently used) protocol. EvoPriority with TimeFitness uncovered slightly

more violations than RandomPriority, while ProposalFitness performed slightly worse, but these differences were not statistically significant. Overall, the results suggest that priority-based testing is capable of finding injected bugs, but may be difficult to guide effectively using evolutionary algorithms and limited in its ability to uncover deeper or unknown bugs.

This paper makes the following contributions:

- We introduce and empirically evaluate EvoPriority, our priority-based evolutionary testing strategy, assessing the impact of different fitness functions on its effectiveness in uncovering consensus violations.
- We extend the Rocket framework by implementing support for priority-based testing and evolutionary search, including the integration of fitness functions.
- We provide a complete replication package, including our implementation, test logs, and experiment configurations, to support reproducibility and future research [5].

2 BACKGROUND INFORMATION AND RELATED WORK

In this section, we give an overview of the XRPL CP, explaining how it processes transactions and reaches agreement across distributed nodes. We also introduce evolutionary testing and explain why it's useful for exploring different message interleavings. Lastly, we look at related work on testing distributed systems, including methods that have been applied specifically to the XRPL CP.

2.1 The XRP Ledger Consensus Protocol

The XRPL CP is a Byzantine Fault Tolerant (BFT) consensus protocol, meaning it can still reach agreement within the network even when some of the nodes are faulty or malicious. This is because the protocol follows the design of the Practical Byzantine Fault Tolerance (PBFT) algorithm [2]. However, unlike PBFT, the XRPL CP allows open membership, by not having a fixed set of trusted validators. To achieve this, each node in the network maintains a list called the Unique Nodes List (UNL), which contains other nodes that it individually trusts [3]. This allows nodes to join or leave the network without requiring global agreement on membership.

Each consensus round, i.e. when the protocol is applied, follows a number of phases [3]:

(1) **Collection Phase:**
Each node collects all valid transactions it has received up to the start of the current consensus round. These may include newly submitted transactions as well as transactions left over from previous consensus rounds. The node sends the collected transactions to the nodes in its UNL, then proceeds to the next phase.

(2) **Proposal Phase:**
The goal of this phase is to reach consensus on a set of transactions which will then be added to the next version of the ledger. At the beginning of the phase, each node sends a proposal to the nodes in its UNL. This proposal contains the set of transactions that the node believes should be applied to the ledger, based on the transactions it collected during the Collection phase.
Each node then compares proposals it received against its own, after which it flags transactions which do not appear in all proposals as disputed transactions.
Nodes continually update their own proposals, adding disputed transactions if they are supported by a sufficient portion of their UNL and removing transactions if they are not widely supported. Initially the threshold to add a transaction to the proposal is set to 50%, after which it increases gradually until it reaches 95%. We refer to this dynamic threshold as the **Proposal Avalanche Threshold**. This is done to bring the proposals of each node closer to each other gradually, reducing the risk of getting stuck in a continuous disagreement.
Nodes have a limited amount of time to reach consensus on a set of transactions. If $\geq$ 80% of the node's UNL agrees on the same transaction set before the timeout, the node considers consensus successful and moves on to the next phase. We refer to this fixed 80% agreement requirement as the **Transaction Agreement Threshold**. Otherwise, the round is considered unsuccessful, and the protocol tries again from the collection phase.

(3) **Validation Phase:**
Nodes finalize the agreed ledger from the proposal phase and broadcast its hash to the network. Once a node receives the same ledger hash from $\geq$ 80% of its UNL, the ledger is considered validated and transactions are applied. This final 80% threshold is referred to as the **Ledger Validation Threshold**.

In order for the XRPL CP to reach agreement and maintain its Byzantine fault tolerance guarantees, some conditions have to be met. Firstly, the protocol requires that at least 80% of the nodes in a given node's UNL behave honestly. Secondly, the protocol relies on a synchronous transition between protocol phases, achieved by predefined time intervals for each phase. This helps the protocol to avoid getting stuck at a certain point and also makes phase transitions more predictable. However, there are some drawbacks, such as possible network latency, which could cause desynchronisation of nodes if not properly handled.

The XRPL CP ensures the following core distributed consensus guarantees [18]:

(1) **Agreement:** Honest nodes do not finalize different ledgers.
(2) **Validity:** Finalized transactions must originate from honest proposals.
(3) **Integrity:** A node cannot finalize more than one ledger per consensus round.
(4) **Termination:** Every correct node eventually finalizes a ledger, assuming sufficient honest participation and synchrony.

2.2 Evolutionary Testing

Evolutionary testing is a software testing technique that automatically generates and evolves test cases over multiple generations. The goal is to explore a diverse and meaningful space of executions to uncover subtle bugs.

Evolutionary testing follows the next steps [1]:

(1) **Initialization:** Randomly generate a initial population of test cases.
(2) **Execution:** Test cases are executed, which provides execution information for the selection step.
(3) **Selection:** Rank test cases with the help of fitness functions and select the top test cases based on their rank.
(4) **Reproduction:** Combine and possibly mutate selected test cases to produce a new set. Go back to execution step with newly generated test cases.

To help guide the evolutionary algorithm to produce meaningful test cases, we use fitness functions. These quantitatively asses the execution of the algorithm being tested on different aspects. In the case of the XRPL CP, some examples may include the duration of the execution or the number of proposals needed to reach consensus on a set of transactions.

2.3 Related Work

A wide range of approaches have been proposed for testing distributed systems, with some tools aiming to systematically explore all possible interleavings of concurrent events. However, this approach quickly becomes impractical for large-scale systems like blockchains. Some examples include dBug [17], DeMeter [11], SAMC [12], and FlyMC [13]. Partial order reduction can be applied to systematic testing approaches, in order to prune redundant interleavings and make systematic exploration more manageable. Still, this fails to make systematic testing a viable option for large-scale applications.

A scalable alternative to systematic testing is randomized testing, which simulates random network partitions and injects consistency-related faults to reveal bugs [14], [4]. Building upon this idea, other techniques introduce enhancements such as reinforcement learning to steer test generation toward buggy executions.

While these concurrency testing approaches have shown success in uncovering bugs in distributed systems, they are not specifically applied to consensus protocols. ByzzFuzz introduces a randomized testing strategy designed specifically for Byzantine fault-tolerant algorithms. It introduces mutations to messages in the network to simulate Byzantine faults. Through this approach, ByzzFuzz has successfully uncovered implementation-level violations [19].

Another approach to concurrency testing involves the use of evolutionary algorithms, as shown in prior work on the XRPL CP consensus protocol [18]. Evolutionary methods were applied to delay-based testing, successfully uncovering a previously unknown bug in the implementation. Building on this idea, our work introduces EvoPriority, which adapts evolutionary testing to a priority-based message scheduling framework. We investigate whether different fitness functions can effectively guide the evolutionary search toward revealing buggy executions in the XRPL CP.

3 METHODOLOGY

This chapter describes the methodology used to generate, execute, and evaluate test cases targeting the XRPL CP. We first detail what inputs a priority-based test needs and how messages are scheduled and dispatched over the network according to their assigned priorities. We then show how this strategy can be extended with evolutionary testing and fitness functions to explore different message interleavings in a targeted way, as to reach scenarios that are more likely to cause bugs.

3.1 Priority-based Testing

In this approach, we control the delivery order of messages by assigning priorities to specific message types. Each test case is defined by a list of integers, referred to as an encoding, which maps priority values to selected message types exchanged between every pair of nodes in the network. This encoding is the input of the priority-based testing strategy and directly influences the behaviour of the consensus process. Rather than assigning priorities to individual messages, we assign them based on their message type. This significantly reduces the size of the search space and keeps the results more interpretable.

Out of all message types in the XRPL CP, only a subset are actually assigned priorities, while the rest are passed through immediately. We refer to this selected subset as core messages, as they represent the fundamental interactions that directly affect the outcome of consensus. By focusing only on these core messages, the strategy targets the areas where concurrency and message ordering are most critical to correctness. This selective prioritization not only reduces the computational cost of each test run but also ensures that the exploration remains focused on meaningful interleavings. Table 1 provides more details about the prioritized message types and their roles in the consensus protocol. Figure 1(a) illustrates how priority values are assigned to different XRPL CP message types. Each coloured block represents a core message type, with the corresponding priority value shown above. Lower values indicate higher delivery priority. For example, TMValidation (TMV) has the highest priority (1), while TMStatusChange (TMSC) has the lowest (7). These priority values determine the order in which messages are dispatched during testing.

To implement this strategy, all network messages are first intercepted. If the intercepted message is not a core message, it is forwarded immediately to its intended destination. Otherwise, the message is matched to its corresponding priority in the encoding and placed into a priority queue. Message dispatch from the queue is controlled by an adaptive sending rate. When the queue grows beyond a target threshold, the dispatch rate is increased to prevent congestion. Conversely, when the queue is too small, the dispatch rate is decreased.

Let I denote the current queue size and r the current dispatch rate in packets per second. The system adjusts r based on how I compares to a defined target queue size T. The update rule is defined as:

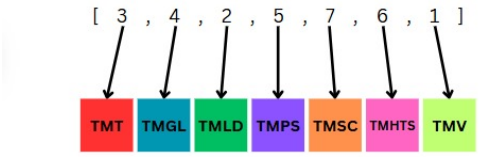
$$r \leftarrow \begin{cases} \min(r \cdot s, r_{\max}) & \text{if } I > T \cdot \alpha \\ \max\left(\frac{r}{s}, \frac{r_{\max}}{6}\right) & \text{if } I < T \cdot \beta \\ r & \text{otherwise} \end{cases}$$

Here, s is a sensitivity ratio controlling how aggressively r is adjusted, and $r_{\max}$ is an upper limit on the dispatch rate. The effective message dispatch rate is then given by:

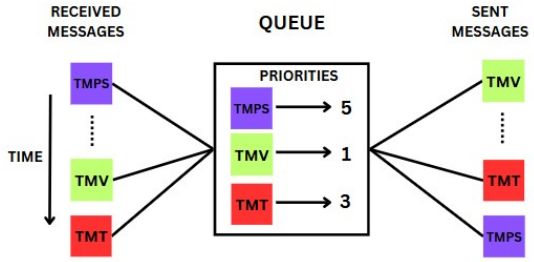
$$\text{packets\_per\_second} = \max(r_{\min}, \lfloor r \rfloor)$$

Table 1: Message Types Assigned Priorities in Priority-Based Testing

Message Type	Description
TMTransaction	Carries client-submitted transactions to be proposed for inclusion in the ledger.
TMGetLedger	Requests a specific ledger or its metadata from a peer.
TMLedgerData	Responds to TMGetLedger with ledger content such as transaction sets or state data.
TMProposeSet	Communicates a validator's current proposal for the next ledger's transaction set.
TMStatusChange	Signals a change in a validator's consensus status (e.g., entering deliberation).
TMHaveTransactionSet	Informs peers that the validator has received a specific transaction set.
TMValidation	Sends a signed validation message for a ledger candidate after consensus.



(a) Mapping priorities to core message types.



(b) Priority-based dispatch ordering from the queue.

Figure 1: Illustration of priority assignment and message dispatch in the priority-based strategy.

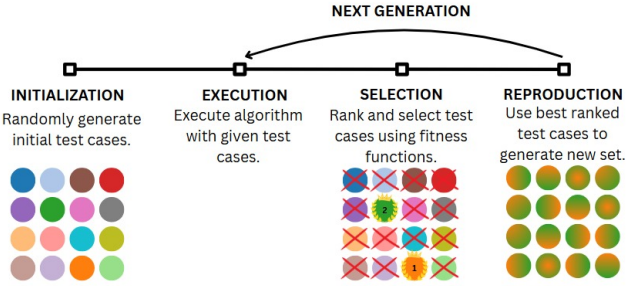


Figure 2: Overview of the evolutionary testing cycle.

where r_{min} is the minimum number of packets that can be dispatched per second. This mechanism allows the queue to gather messages to make more use of the assigned priorities, while adjusting the dispatch rate to prevent build-up and maintain a steady throughput of messages over the network. Figure 1(b) illustrates how messages are intercepted, assigned priorities and after dispatched in order of these priorities. For example, TMProposeSet (TMPS) is intercepted before TMValidation (TMV), but assigned a lower priority, so it is dispatched last.

3.2 EvoPriority Approach and Fitness Functions

Building on the priority-based testing method, we apply an evolutionary algorithm to explore the space of possible message interleavings. In this approach, each population is defined by an encoding, a list of integers used to assign priority values to core messages. Populations evolve over generations and contain a number of test cases. The goal is to discover message schedules which are more likely to expose bugs in the XRPL CP, guided by the help of fitness functions.

The evolutionary process follows four main stages: initialization, execution, selection, and reproduction. As illustrated in Figure 2, the process begins by randomly generating a set of test cases. Each test case is executed and then evaluated using a fitness function:

(1) **TimeFitness**: Measures the average time taken for the XRPL CP to reach consensus. The more time a test case takes to execute, the higher the score it receives. The intuition behind this is that longer runtimes may lead to delayed consensus progress and violations of the termination property. TimeFitness is defined as:

$$\text{TimeFitness}(T) = \frac{1}{n} \sum_{i=1}^{n} \text{validation\_time}_i \tag{1}$$

Where:

- T is the test case (i.e. encoding),
- n is the number of validator nodes,
- validation_time_i is the time taken by node i to validate a ledger.

(2) **ProposalFitness**: Rewards test cases in which nodes send more proposal messages in a single consensus round. This indicates increased difficulty in reaching agreement. ProposalFitness is defined as:

$$\text{ProposalFitness}(T) = \sum_{i=1}^{n} \text{count}_{\texttt{ProposeSet},i} \tag{2}$$

Where:

- $\text{count}_{\texttt{ProposeSet},i}$ is the number of `ProposeSet` messages sent by node i.

To evolve test cases, we use a multi-objective genetic algorithm based on NSGA-II. Each test case is evaluated according to two objectives:

- **Execution Fitness:** calculated using either TimeFitness or ProposalFitness.
- **Consensus Violations:** Number of violations found during execution.

Each generation proceeds as follows. First, parent individuals are selected using tournament selection based on dominance (DCD), which considers both objectives. Two selected parents are recombined using Simulated Binary Crossover (SBX) [7], producing two offspring. These offspring are then possibly mutated using Gaussian mutation [8], with each gene having a small independent mutation probability. All genes are clamped to valid bounds after mutation.

To maintain diversity and select the next generation, we apply NSGA-II [9], which sorts individuals into Pareto fronts and applies crowding distance to break ties.

4 EVALUATION

In this section, we evaluate the effectiveness of the *EvoPriority* approach by examining how different fitness functions guide the evolutionary search toward consensus violations in the XRPL CP. We begin by listing our research questions. Next, we detail the experimental setup, including the testing environment, key parameters, and metrics used. Finally, we present and discuss the results of our experiments, comparing the ability of each fitness function to uncover violations and analysing their impact on search efficiency.

4.1 Research Questions

We test the XRPL CP in order to answer the following questions:

- **RQ1:** *Can RandomPriority and EvoPriority uncover bugs in the XRP Ledger Consensus Protocol?*
  This question investigates the bug finding capability of our priority-based evolutionary algorithm in the XRPL Consensus Protocol.
- **RQ2:** *How does the bug detection performance of EvoPriority compare to the RandomPriority strategy?*
  This question investigates the added value of using fitness functions to guide the evolutionary search, compared to a baseline approach that explores the space randomly.
- **RQ3:** *How does the choice of fitness function influence the effectiveness of EvoPriority?*
  This question investigates how different fitness functions affect the total number violations found.

4.2 Experimental Setup

We tested the XRPL CP using the Rocket testing framework [15]. The network consists of seven validator nodes running in Docker containers, where each node includes all other nodes in its UNL, forming a fully connected trust model. Experiments were conducted on a dedicated research server equipped with 2× AMD EPYC 7H12 64-core CPUs (128 cores/256 threads total) and 256 GB of RAM.

To initialize the ledger state, we submit three genesis transactions that fund three accounts (Accounts 1, 2, and 3). The test concurrently issues four regular transactions from Account 1, which attempts to overspend its balance by transferring funds to Accounts 2 and 3 through four different validator nodes. The transactions are:

- Tx1 = {Account1 ⟶ Account2, 80 XRP}
- Tx2 = {Account1 ⟶ Account3, 81 XRP}
- Tx3 = {Account1 ⟶ Account3, 82 XRP}
- Tx4 = {Account1 ⟶ Account2, 83 XRP}

All four transactions are submitted simultaneously 2 seconds after the start of execution to nodes 0, 1, 5, and 6 respectively.

Table 2: Values Used for Priority-Based Dispatch Parameters

Parameter	Value
`target_inbox`	10
`overflow_factor`	1.2
`underflow_factor`	0.8
`sensitivity_ratio`	1.2
`min_packets_per_second`	100
`max_events`	1000

A single test case consists of submitting these transactions to the network and observing the outcome over 14 ledgers. For the first 10 ledgers, we apply a priority-based strategy to control the delivery order of consensus messages. For the final 4 ledgers, we disable message manipulation to observe whether the network can recover and reach agreement naturally.

In our priority-based strategy, the key parameters controlling the rate at which messages are sent over the network are all mentioned in Table 2.

To answer the research questions, we run concurrency tests on two different versions of the XRPL CP, a bug seeded version and the original current version used by the XRPL (2.4.0):

- **Bug-Seeded Version:** A modified version of the original source code of the XRPL CP, which lowers the **Transaction Agreement Threshold** and the **Ledger Validation Threshold**, from 80%, down to 40%. The **Proposal Avalanche Threshold** is also fixed to 40% instead of gradually increasing over each consensus round. These modifications create more opportunities for nodes to agree on two different ledgers, causing a split in the network.
- **Original Version:** An unmodified version of the currently used XRPL CP source code (2.4.0).

For each XRPL CP version that we used, we applied the following priority-based testing strategies:

- **RandomPriority:** generates a random list of priorities for each test case.
- **EvoPriority (using TimeFitness):** the priority-based evolutionary testing guided with the TimeFitness fitness function.
- **EvoPriority (using ProposalFitness):** the priority-based evolutionary testing guided with the ProposalFitness fitness function.

Each configuration of different XRPL CP version and priority-based testing strategy (as listed in Table 3), is executed over of 50 generations with populations of size 10. This means that a generation executes 10 test cases with 10 different encodings, each of which is executed once (1 iteration). For the RandomPriority

Table 3: Experimental Configurations

Config	XRPL CP Version	Testing Strategy
C1	Bug-seeded	RandomPriority
C2	Bug-seeded	EvoPriority (TimeFitness)
C3	Bug-seeded	EvoPriority (ProposalFitness)
C4	Original	RandomPriority
C5	Original	EvoPriority (TimeFitness)
C6	Original	EvoPriority (ProposalFitness)

strategy, all encodings are created randomly using different seeds. As for the EvoPriority strategies, they begin with a randomly generated initial generation, after which selection is applied, using their respective fitness functions. The best performing test case encodings are selected and used to generate the next generation of test cases.

4.3 Results

Each test run configuration output logs containing information about the consensus protocol and evolutionary algorithm executions. From these logs we were able to determine whether or not violations of the XRPL CP occurred. We checked two types of violations [3]:

- **Liveness Violations:** a violation of the termination property. For this type of violation we check if ledgers are finalized in a specific time frame. This time frame is set to 65 seconds. Usually, the XRPL CP will validated a ledger every four to five seconds, hinting at a problem if it is unable to do so in 65 instead.
- **Safety Violations:** a violation of the agreement, validity or integrity properties. For this type of violation we check if: (1) two nodes have both validated two different ledgers, (2) a node finalized more than one ledger in a consensus round.

Table 4 presents the total number of violations identified by each testing configuration across 50 generations, with 10 test cases evaluated per generation. Since no violations were detected on the original version of the XRPL consensus protocol under any strategy, statistical analysis and visualization were limited to the bug-seeded version only. The cumulative violations observed for each strategy on the bug-seeded version are shown in Figure 3.

To assess whether the differences in violations detected by the strategies are statistically significant, we performed independent two-sample t-tests on the bug-seeded results. These tests evaluate whether the mean number of violations per generation differs between strategies under the assumption of normality. The resulting p-values are reported in Table 5. A p-value quantifies the probability of observing a difference at least as large as the one measured, assuming no true difference exists. A low p-value (typically below 0.05) indicates that such a difference is unlikely to have occurred by chance and is considered statistically significant.

Based on these results from 500 test cases performed on each configuration, we can discuss our research questions:

(RQ1) Can RandomPriority and EvoPriority uncover bugs in the XRP Ledger Consensus Protocol?

Table 4: Number of violations found for each testing configuration

Config	Liveness Violations	Safety Violations	Total Violations
C1	2	7	**9**
C2	3	9	**12**
C3	4	3	**7**
C4	0	0	**0**
C5	0	0	**0**
C6	0	0	**0**

Table 5: p-values from independent t-tests comparing violations per generation between strategies (bug-seeded version)

	Random Priority	Time Fitness	Proposal Fitness
Bug-Seeded Version			
TimeFitness	0.419	-	0.306
ProposalFitness	0.809	0.306	-

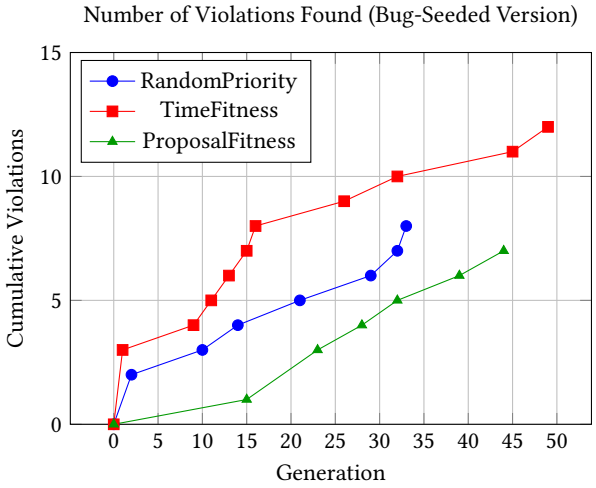


Figure 3: Cumulative violations found up to each generation on the bug-seeded version of the XRPL CP

- **Observed Results:** All three testing strategies were able to uncover violations in the bug-seeded version of the XRPL consensus protocol, with total counts ranging from 7 to 12 violations (Table 4). In contrast, none of the strategies found any violations in the original (non-seeded) version across all 50 generations and test cases.
- **Discussion:** The small number of violations found over the 50 generations might indicate that priority-based testing, regardless of whether it is guided by fitness or random selection, has limited effectiveness in discovering bugs under the current experimental conditions.

These results suggest that priority-based testing is capable of detecting bugs across different strategies. Both RandomPriority and EvoPriority successfully uncovered violations in the bug-seeded version of the XRPL consensus protocol. However, the overall number of violations is relatively low, and no bugs were uncovered on the original version. Taken together, these findings provide a answer to our first research question (RQ1). On the bug seeded version, the answer is yes, but on the original source code, no violations were observed with the current experimental setup.

(RQ2) How does the bug detection performance of EvoPriority compare to the RandomPriority strategy?

- **Observed Results:** From Table 4, we see that EvoPriority using TimeFitness outperformed RandomPriority, but ProposalFitness did not. The ranking of total violations found over the 50 generations is: EvoPriority (TimeFitness), followed by RandomPriority, and then EvoPriority (ProposalFitness). None of the strategies found any violations on the original version of the XRPL CP, showing no observable difference in their performance under these conditions.
  However, Table 5 shows that none of the differences observed in the bug-seeded configurations are statistically significant. This indicates that the higher violation count for EvoPriority (TimeFitness) and the lower count for EvoPriority (ProposalFitness) may be due to random variation rather than a consistent effect of the fitness functions. These results suggest that EvoPriority using TimeFitness, does not outperform a random search of the encoding space in a statistically meaningful way, answering our second research question (RQ2).
- **Discussion:** The lack of a significant improvement from the EvoPriority strategy could have two underlying causes. First, the nature of the priority-based approach itself may contribute to the instability. Small changes to an encoding can result in disproportionately large changes in system behaviour, due to the message timing and ordering. This sensitivity means that even similar encodings can produce widely different execution paths, reducing the evolutionary process to what is effectively a random search and limiting its ability to converge toward more effective test cases over time. In Figure 3 we can see that all strategies have very similar bug finding capabilities, finding bugs gradually across generations, further showing how EvoPriority struggled to guide towards bug-inducing executions.
  Second, each encoding in the evolutionary process was only executed once, meaning the fitness score is based on a single observation. This introduces a high degree of noise, possibly making it difficult for the evolutionary algorithm to reliably assess the quality of a given encoding and guide selection effectively.

(RQ3) How does the choice of fitness function influence the effectiveness of EvoPriority?

- **Observed results:** From Table 4 and Table 5 we can observe that even though EvoPriority performed better using TimeFitness than it did using ProposalFitness, these differences are most likely caused by a random variation, indicated by the p-value between the two strategies in Table 5.
  To further compare the performance of EvoPriority with both fitness functions, we can see in Figure 3 that their bug finding capabilities on the seeded version are linear. This means that both fitness functions lack the ability to guide EvoPriority in creating more bug-inducing executions.
- **Discussion:** If the fitness functions were to actually help guide EvoPriority to finding more bugs, we could have expected an increase in the number of bugs found as the algorithm reached higher generations. There is nothing to suggest this behaviour in our results. This could possibly be explained by the same reasons to why EvoPriority didn't significantly outperform RandomPriority: the small number of iterations per test case or the large change in behaviour caused by small changes to encodings.
  These results answer our third and final research question (RQ3), leading us to the conclusion that, under these testing configurations, the choice of fitness function does not influence the bug finding performance of EvoPriority.

5 RESPONSIBLE RESEARCH

We considered both the reproducibility of the experimental methods and the ethical implications of testing the XRPL CP.

5.1 Reproducibility

To ensure reproducibility, we provide a replication package [5] that contains:

- The exact repository configurations used during testing,
- The result logs of all test runs,
- A README file with instructions on how to reproduce the experiments.

Each test execution can be reproduced by following the documented configuration steps and re-running the associated strategy within the Rocket framework.

5.2 Ethical Considerations

Although the XRPL CP is a production-grade protocol powering financial infrastructure, all experiments in this study were conducted on an isolated, private test network. No tests were run on the live XRP Ledger network. There was no risk of disrupting real-world financial transactions or affecting users' assets.

If any previously unknown bugs or vulnerabilities had been discovered during testing, we were prepared to follow a responsible disclosure protocol. This ensures that any findings would be reported confidentially to the relevant maintainers of the XRPL software, providing them the opportunity to address the issue before public disclosure.

Our work aligns with the broader goal of strengthening the reliability and fault-tolerance of blockchain systems through rigorous and ethical testing practices.

6 THREATS TO VALIDITY

One limitation of this evaluation is that it focuses exclusively on the XRP Ledger Consensus Protocol. As a result, the findings may

not generalize to other distributed consensus algorithms with different communication models, fault assumptions, or consensus mechanisms. The effectiveness of priority-based evolutionary testing and the selected fitness functions could vary significantly in other systems.

Another potential threat is the inherent nondeterminism in the test environment, including message scheduling, timing variations, and random elements in the evolutionary process. This can introduce variability in the outcomes across runs. To reduce this risk, we executed a large number of generations and test cases per strategy, allowing consistent patterns and trends to emerge despite the stochastic nature of the setup.

7 CONCLUSION AND FUTURE WORK

In this paper, we investigated the effectiveness of the EvoPriority strategy, an evolutionary algorithm based on priority-based event representation, for testing the XRP Ledger Consensus Protocol. Specifically, we evaluated how different fitness functions (TimeFitness and ProposalFitness), influence EvoPriority's ability to uncover protocol violations. Our results show that EvoPriority is capable of detecting faults when seeded bugs are present, identifying 12 violations using TimeFitness and 7 using ProposalFitness. These findings confirm that the evolutionary strategy is functional.

However, when comparing EvoPriority against a random priority baseline, we observed that EvoPriority did not make a statistically significant improvement in bug detection performance, regardless of the fitness function used. The comparison between TimeFitness and ProposalFitness revealed no meaningful difference in their effectiveness. These results suggest that, despite its theoretical advantages, the evolutionary approach may not be well suited for priority-based testing. A likely explanation lies in the instability of the priority-based encodings, meaning that small mutations in the priorities can result in drastic changes in message delivery order, leading to unpredictable consensus protocol behaviour. This pairs badly with the evolutionary algorithm's ability to exploit local fitness gradients, effectively reducing its search to a randomized process.

In conclusion, while EvoPriority demonstrates that it can be applied to the XRP Ledger Consensus Protocol, our findings raise concerns about the suitability of priority-based encodings for evolutionary search and the ability of fitness functions to guide the search towards bug-inducing executions.

Future work should aim to address some of the limitations imposed by the constrained timeline of this study. We were limited to running a single test iteration per encoding, which may have reduced the ability for our fitness functions to properly guide EvoPriority and might of contributed to the observed lack of statistically significant differences. Time for test runs with more iterations per test case could provide a more reliable picture of the algorithm's performance and sensitivity to different configurations. Evaluating additional fitness functions may help determine whether the lack of guidance observed in this study was due to the fitness functions themselves or the unpredictability of the priority-based encodings. It would also be valuable to test other message scheduling strategies with evolutionary algorithms, such as delay-based encodings, which may be easier to guide in the search for bugs. Finally, extending the testing framework to additional systems similar to the XRP Ledger Consensus Protocol. This could help generalize the findings and explore whether the limitations of EvoPriority are system-specific or more broadly applicable across Byzantine fault-tolerant protocols.

REFERENCES

[1] Thomas Bäck and Hans-Paul Schwefel. 1993. An Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation* 1, 1 (1993), 1–23. https://doi.org/10.1162/evco.1993.1.1.1

[2] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI '99)*. USENIX Association, New Orleans, Louisiana, USA, 173–186. https://www.usenix.org/conference/osdi-99/practical-byzantine-fault-tolerance Also described in Castro's MIT thesis, 1999.

[3] Bradley Chase and Ethan MacBrough. 2018. Analysis of the XRP Ledger Consensus Protocol. *arXiv preprint arXiv:1802.07242* (2018). https://arxiv.org/abs/1802.07242

[4] Hongxu Chen, Weiyi Dou, Dinghao 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)*. IEEE, Melbourne, Australia, 536–547. https://doi.org/10.1145/3324884.3416600

[5] Călin Ciocănea. 2025. Replication Package - EvoPriority: Evaluating Fitness Functions in Priority-Based Evolutionary Testing for the XRP Ledger Consensus Protocol. https://doi.org/10.5281/zenodo.15715119 Version 1.0.

[6] CoinStats. 2025. XRP (Ripple) Live Price, Market Cap and Charts. https://coinstats.app/coins/ripple/. Accessed: 2025-06-03.

[7] Kalyanmoy Deb and Ram Bhusan Agrawal. 1995. Simulated Binary Crossover for Continuous Search Space. *Complex Systems* 9, 2 (1995), 115–148.

[8] Kalyanmoy Deb and D. Deb. 2014. Analysing Mutation Schemes for Real-Parameter Genetic Algorithms. *International Journal of Artificial Intelligence and Soft Computing* 4, 1 (2014), 1–28.

[9] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (April 2002), 182–197. https://doi.org/10.1109/4235.996017

[10] Golden. 2025. Ripple (cryptocurrency). Golden.com. https://golden.com/wiki/Ripple_%28cryptocurrency%29-YJG "To date, more than 300 financial institutions have joined the Ripple initiative worldwide.".

[11] Hui Guo, Minhui Wu, Lidong Zhou, Guangyu Hu, Junfeng Yang, and Lidong Zhang. 2011. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Tom Wobber and Peter Druschel (Eds.). ACM, Cascais, Portugal, 265–278. https://doi.org/10.1145/2043556.2043584

[12] 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)*. USENIX Association, Broomfield, CO, USA, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

[13] Jeffrey F. Lukman, Heng Ke, Cristian A. Stuardo, Raka O. Suminto, David H. Kurniawan, Daniel Simon, Samuel Priambada, Chen Tian, Fan Ye, Tanakorn Leesatapornwongsa, Abhishek Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, Dresden, Germany, 20:1–20:16. https://doi.org/10.1145/3302424.3303965

[14] Rupak Majumdar and Filip Niksic. 2018. Why is Random Testing Effective for Partition Tolerance Bugs? *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 46:1–46:24. https://doi.org/10.1145/3158114

[15] Burcu Kulahcioglu Ozkan and Annibale Panichella et al. 2025. Rocket: A System-Level Fuzz Testing Framework for the XRPL Consensus Algorithm. https://conf.researchr.org/details/icst-2025/icst-2025-testing-tool-data-showcase/3/Rocket-A-System-Level-Fuzz-Testing-Framework-for-the-XRPL-Consensus-Algorithm. Presented at ICST 2025 Testing Tool Data Showcase.

[16] David Schwartz, Noah Youngs, and Arthur Britto. 2014. *The Ripple Protocol Consensus Algorithm.* Technical Report. Ripple Labs Inc. https://www.xrpchat.com/topic/1351-the-ripple-protocol-consensus-algorithm/ Preliminary whitepaper, not reflecting the current state of the protocol..

[17] Jiri Simsa, Randal Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *5th International Workshop on Systems Software Verification (SSV'10)*. USENIX Association, Vancouver, BC, Canada. http://www.cs.cmu.edu/~jsimsa/dbug

[18] M. van Meerten, B. K. Ozkan, and A. Panichella. 2023. Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm. In *2023*

IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, 36–47. https://doi.org/10.1109/ICSE-SEIP58662.2023.00015

[19] Lars Winter, Florian Buşe, Daniël de Graaf, Kai von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. Randomized testing of byzantine fault tolerant algorithms. *Proceedings of the ACM on Programming Languages* 7, OOPSLA1 (2023), 757–788. https://doi.org/10.1145/3571205