May the Delays Be Ever in Your Favor: Genetic Operators in Delay-Based Testing of the XRPL Consensus Algorithm

Wishaal Kanhai[1]

Supervisor(s): Burcu Kulahcioglu Ozkan[1]**, Annibale Panichella**[1]**, Mitchell Olsthoorn**[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 21, 2025

Name of the student: Wishaal Kanhai
Final project course: CSE3000 Research Project
Thesis committee: Burcu Kulahcioglu Ozkan, Annibale Panichella, Mitchell Olsthoorn, Jeremie Decouchant

May the Delays Be Ever in Your Favor: Genetic Operators in Delay-Based Testing of the XRPL Consensus Algorithm

Wishaal Kanhai*

Delft University of Technology
Delft, The Netherlands

Abstract

The XRP Ledger (XRPL) relies on a Byzantine fault-tolerant consensus algorithm to ensure global agreement on transactions across distributed nodes. Despite its critical financial role, the implementation remains under-tested. While prior work has shown the potential of evolutionary testing to uncover potential consensus violations in XRPL, the role of genetic operator selection in this process remains unexplored. We address this research gap by presenting a comparative evaluation of four evolutionary configurations that differ in their balance of exploration and exploitation. The system is tested by injecting network delays to simulate adverse conditions and trigger violations. Our results show that the balance of exploration and exploitation affects the performance of bug detection: configurations that favor exploitation, complemented by subtle exploration, yield the most favorable results. In addition, we contribute an extensible testing method tailored to XRPL but applicable to other distributed systems.

Keywords

XRPL, Consensus Algorithms, Consensus Protocols, Byzantine fault-tolerance, Evolutionary Testing, Genetic Operators, Message Delays, Message reordering

1 Introduction

The XRP Ledger (XRPL) consensus algorithm, commonly referred to as a protocol, is designed to be Byzantine fault-tolerant (BFT), enabling the decentralized XRPL system to reach agreement on a single correct value, even when a subset of processes behaves maliciously or unpredictably [1], [2]. The XRP Ledger[1] is a public, decentralized blockchain developed by Ripple to facilitate efficient cross-border transactions using the XRP cryptocurrency. XRPL processes hundreds of millions of of dollars in transactions daily, making the reliability of its consensus implementation critically important.

Although the algorithm offers strong theoretical guarantees, its practical implementation remains susceptible to human errors [3], [4], which could lead to critical bugs. Such bugs may allow attackers to halt system progress or even validate invalid transactions. Consequently, rigorous testing of complex systems such as XRPL is essential to ensure both their correctness and robustness in practice.

XRPL validators maintain copies of the ledger history and coordinate through a proposal-based process where they exchange transaction sets until sufficient agreement is reached to finalize a new ledger. Testing such decentralized systems presents significant challenges due to the nondeterministic nature of message delivery between distributed nodes, potential network delays, and the complex interactions that can lead to subtle concurrency bugs that are difficult to reproduce and detect in traditional testing environments [4].

Previous testing efforts have effectively uncovered vulnerabilities in the XRPL consensus algorithm. An evolutionary testing approach was introduced that enables message reordering via delay injection and scheduling prioritization [4], leading to the discovery of a previously unknown concurrency bug. Building on this, the ByzzFuzz algorithm [3] applied message mutations and simulated network faults, revealing a critical bug capable of halting system progress entirely. These findings led to the development of Rocket [5], a comprehensive testing framework designed to simulate an entire network of XRPL nodes and inject faults based on user-defined test cases.

Despite these advancements, system-level testing of blockchain consensus algorithms, including XRPL, remains underexplored [6], [3]. While prior work has applied evolutionary testing with delay injection to uncover concurrency bugs [4], the impact of genetic operator selection has yet to be evaluated. These operators differ in their degrees of exploration and exploitation, and this choice can play a critical role in determining how effectively bugs are uncovered during testing [7].

In this paper, we evaluate how different configurations of genetic operators influence the effectiveness of evolutionary testing for the XRPL consensus algorithm using delay-based event representations. We present a method that evolves test cases consisting of sequences of message delays, which are applied to consensus messages within a simulated XRPL network. By varying the genetic operator configurations, we explore how different balances of exploration and exploitation impact the discovery of edge-case behaviors and potential vulnerabilities. Specifically, we experiment with two crossover operators: Simulated Binary Crossover (SBX), which emphasizes exploitation, and Blend-α crossover, which favors exploration. We also consider two mutation operators, with polynomial mutation favoring exploitation and Gaussian mutation favoring exploration.

Our experimental results show that a configuration combining exploitation-oriented crossover using SBX with exploration-focused mutation through Gaussian mutation performs particularly well. This setup is especially effective in discovering local optima within the input space, leading to executions that are more prone to violations and enabling the efficient discovery of bugs in the XRPL consensus algorithm.

[1]https://xrpl.org/

These findings suggest that the XRPL input space exhibits locally dense but globally sparse distributions of bug-revealing inputs. In such a landscape, fine-grained exploitation is key to exposing bugs once a promising region is found, while periodic exploration is necessary to escape local optima and uncover new fault clusters.

At the same time, unguided randomized testing still performs competitively, often generating a broader and more diverse set of inputs as it explores the input space without directional constraints. Depending on the testing objective, whether it is maximizing diversity or intensifying bug discovery, this diversity may be either advantageous or less effective compared to guided evolutionary methods.

This paper makes two primary contributions to the field of concurrency testing in distributed systems:

- An in-depth evaluation of different genetic operators in evolutionary testing of the XRPL consensus algorithm.
- A flexible and extensible testing method that supports additional configuration parameters to guide the generation of test cases.

2 Background and Related Work

This section outlines core functional aspects (Section 2.1) and correctness properties (Section 2.2) of the XRPL consensus algorithm. Furthermore, it reviews related work in the field of testing distributed systems (Section 2.3) and outlines the application of evolutionary testing within the context of the XRPL consensus algorithm (Section 2.4).

2.1 Reaching Consensus

The XRPL system utilizes its consensus algorithm to ensure that a distributed set of nodes agrees on a single, validated set of transactions. The nodes that actively participate in this consensus process are also known as *validators* or *validator nodes*. Each validator maintains a record of the *ledger* history, commonly called the *blockchain* in other blockchain systems. Every new ledger contains the set of transactions that have been validated since the previous ledger, along with additional metadata. This mechanism ensures network consistency and prevents double spending, where the same funds could be used in multiple transactions.

Unlike other BFT and blockchain protocols, the XRPL consensus algorithm introduces a unique approach to trust by supporting subjective and asymmetric trust relationships among nodes [8]. The XRPL consensus algorithm allows each validator to rely on a personalized set of trusted peers, which are simply other validators, known as the Unique Node List (UNL). Validators participate in consensus by evaluating proposals only from the nodes included in their respective UNLs. Consensus is strongly guaranteed as long as at least 80% of a validator's UNL behaves honestly, and there is sufficient overlap between the UNLs of different validators to ensure network-wide agreement, roughly at least 90% [2].

Each consensus round in XRPL has three phases: open, proposal, and validation [2]. In the open phase, clients can submit transactions to the network. These transactions are temporarily held by validators and remain pending until they are potentially included in the next validated ledger. During the proposal phase, validators exchange and adjust transaction proposals, resolving disputes with a rising agreement threshold through avalanche tuning. If at least 80% agreement is reached within a validator's UNL, it moves to the validation phase; otherwise, it returns to the open phase. In the validation phase, validators confirm agreement on the ledger and broadcast validation messages. If a node receives matching validations from at least 80% of its UNL, it finalizes the ledger and applies its transactions permanently, ensuring network consistency and integrity.

2.2 Consensus Properties

The correctness of the XRPL consensus algorithm is characterized by the following key properties of BFT consensus theory [9]:

1. **Termination:** Every correct process eventually decides on some value.
2. **Validity:** A correct process may only decide on a value that was proposed by a correct process.
3. **Integrity:** No correct process decides more than once.
4. **Agreement:** No two correct processes decide differently.

In XRPL, the value being decided during consensus is the next ledger to be validated and appended to the ledger history. Validators independently build candidate ledgers from received transactions and exchange proposals to agree on a single correct ledger. The process is asynchronous, and concurrent message delivery causes nondeterminism. Variations in message order, especially with conflicting transactions, can lead to different execution paths. The committed transactions depend on message timing and order. This makes consensus correctness, safety and liveness, critical. Safety properties (agreement, validity, integrity) ensure no divergence, invalid values, or multiple decisions. Liveness (termination) ensures consensus completes so the network stays responsive.

2.3 Related Work

A large variety of approaches have been effective in verifying distributed systems. Systematic methods explore all possible execution orders but do not scale well to large systems. Conversely, randomized testing offers better scalability by producing highly varied executions, often improved with reduction or learning techniques, which has been shown to outperform systematic methods in practice [10]. Some techniques simulate random network partitions to stress distributed systems [11], while others, such as blockchain-specific fuzzing frameworks [12], have uncovered real-world vulnerabilities across multiple platforms. Probabilistic concurrency testing adds theoretical guarantees [13], and other methods enhance test diversity using partial order reduction [14], semantic reduction [15], or reinforcement learning [16]. Manual testing is also commonly employed, but it is labor-intensive and difficult to scale to the complexity of distributed systems [5].

Testing the XRPL consensus algorithm is challenging due to its inherent nondeterminism, which limits the effectiveness of traditional deterministic methods. Even though system-level testing of XRPL remains underexplored, the ByzzFuzz algorithm [3] has shown that randomized testing can effectively verify its consensus protocol. By applying small mutations to valid messages and simulating network failures, ByzzFuzz triggered subtle edge-case executions. These executions revealed bugs linked to weak trust assumptions in UNLs and uncovered a critical implementation flaw that could halt the consensus process.

2.4 Evolutionary Testing

Building on randomized testing, evolutionary testing introduces search guidance to more effectively uncover challenging execution scenarios in nondeterministic systems like XRPL. Such approaches start with a random population of test cases, represented as individuals composed of genes, each gene encoding a specific parameter of the test case. These individuals are evaluated using fitness functions, and the best are selected as parents to produce a population of offspring, new test cases, through crossover and mutation. The fittest individuals from both populations form the next generation, and this process repeats until a stopping condition is met.

Evolutionary testing has been widely used in search-based software testing to automatically generate and refine test cases for specific goals [17]. Driven by fitness functions that assess how well test cases meet those goals, these methods have proven effective across unit, integration, and system testing [18], [19], [20]. They consistently outperform random testing in coverage and bug detection, making them well-suited for complex, large-scale software.

Adapting this strategy to distributed systems, prior work has applied evolutionary testing to uncover concurrency bugs in the XRPL consensus algorithm, showing its effectiveness in real-world distributed systems [4]. However, the impact of different genetic operator configurations in this context remains unexplored. Since these operators control how the search process balances exploration and exploitation, their configuration may influence the effectiveness of bug discovery in XRPL [7].

3 Methodology

We apply evolutionary testing to evaluate how different genetic configurations affect the effectiveness of bug discovery in XRPL. In this section, we elaborate on four key components of our methodology: the use of message delays to trigger edge-case executions (Section 3.1), the application of these delays in a local test network (Section 3.2), the generation and evolution of test cases using different genetic operators (Section 3.3), and the detection of consensus violations (Section 3.4).

3.1 Message Delays and Reordering

In distributed networks, injecting delays alters the timing of message delivery, which can indirectly result in message reordering. One such scenario is illustrated in Figure 1. By introducing a delay to the network event represented by the orange square, a novel message arrival order is produced.

Message reordering in the asynchronous XRPL network can cause conflicting transactions to be committed inconsistently [21], [22], leading to nondeterministic ledger states. By applying evolutionary testing in combination with varying genetic operators, our method efficiently identifies buggy executions without requiring exhaustive exploration of all possible message schedules.

3.2 Controlling a Local XRPL Network

We apply message delays in a local XRPL test network to gain full control over message transmissions. For this purpose, we use the

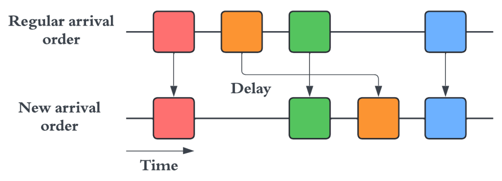

Figure 1: An example message reordering by applying a delay.

Table 1: XRPL consensus-related message types.

Message Type	Description
ProposeSet	Contains a set of transactions proposed by a node for the next ledger.
StatusChange	Signals that a node has closed its current ledger or accepted a new one.
Validation	Confirms a node's agreement on the validity of a new ledger.
Transaction	Carries a client-submitted transaction intended for inclusion.
HaveTransactionSet	Indicates that the sender has obtained a specific transaction set.
GetLedger	Requests missing ledger or transaction data from peers.
LedgerData	Responds to data requests by sending ledger and transaction information.

Rocket framework [5], which allows us to simulate network faults in an isolated XRPL test network. Rocket intercepts all inter-node messages and allows for custom message-handling strategies. In our method, this functionality is used to apply specific delays to messages as defined by our test cases. Additionally, Rocket provides access to detailed runtime logs, which we use to analyze test case performance during post-execution evaluation.

Our method utilizes a network consisting of 5 validator nodes with 100% overlap in their UNLs. This configuration allows us to create adverse but controlled conditions, as demonstrated in previous work [4], while preserving the theoretical guarantees of XRPL consensus under full UNL overlap [2], [23].

The XRPL consensus algorithm relies on a variety of message types exchanged between validators to coordinate agreement on the next ledger. These messages serve distinct roles across the consensus phases [23], including proposing transactions, signaling state changes, validating ledgers, and synchronizing ledger data. Table 1 provides an overview of the message types relevant to the consensus process, such as `ProposeSet`, `StatusChange`, `Validation`, and others involved in transaction propagation and ledger synchronization.

3.3 Evolving Test Cases

Each individual (test case) is represented by an encoding of delays assigned to network events. For every consensus-related event in the network, a corresponding delay is defined. Each event is defined as a message sent over the network, represented by the tuple (sender, receiver, message type).

To evaluate the performance of each individual, we define two fitness functions that guide the search toward executions likely to reveal consensus violations:

f_t: Measures the average time taken to validate one ledger (in seconds). This is motivated by prior work [4], which suggests that longer execution times correlate with conditions under which concurrency bugs are more likely to emerge.

f_v: Counts the number of liveness and safety violations observed during execution.

We use a population of size 10 and evolve it over 50 generations. The choice of a relatively small population size is motivated by existing literature, which shows that large populations do not necessarily improve performance in evolutionary algorithms [24]. Additionally, smaller populations are shown to be particularly well-suited for computationally expensive problems [25], such as ours, where each evaluation involves simulating complete XRPL consensus rounds.

We initialize the first generation with a population of individuals whose gene values, representing message delays, are randomly chosen between 0 and 4000 milliseconds. This range is selected to provide sufficient room for reordering messages within an entire consensus round, while still allowing the XRPL network to make progress (as noted in [4]). In practice, a single consensus round in XRPL typically completes within 3 to 5 seconds. The 4000 ms value also serves as a strict upper bound that is never exceeded during evolution.

Offspring are generated by selecting parents through tournament selection using the crowded comparison operator [26], from which a new population of 10 individuals is produced. The next generation is then formed using the NSGA-II selection procedure, which selects the best 10 individuals from the combined pool of the prior and offspring populations.

An individual in the offspring generation is produced from two parents using a combination of genetic operators. In our evaluation, we consider two crossover operators and two mutation operators, chosen to represent different balances between exploration and exploitation.

We consider the following crossover operators:

- **Simulated Binary Crossover (SBX)** with $\eta = 3$ [27]: This setting favors exploitation by producing offspring near the parent solutions, consistent with the configuration used in prior research [4]. Each gene is recombined with a probability of 0.5, as supported by earlier work [4], [28].
- **Blend-α Crossover** with $\alpha = 0.7$ [29]: This setting favors exploration by generating offspring that can lie beyond the range of the parents, as argued in [30].

We consider the following mutation operators:

- **Polynomial Mutation** with a distribution index of $\eta = 20$, which favors exploitation by producing more localized modifications [31].
- **Gaussian Mutation**, which adds a value sampled from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 40$ [31], [4]. This encourages exploration by enabling broader search steps.

For both mutation operators, the mutation probability is set to $\frac{1}{n}$, where n is the number of genes in an individual. This means that, on average, one gene is mutated per individual, which is a commonly used setting in the literature [4], [31].

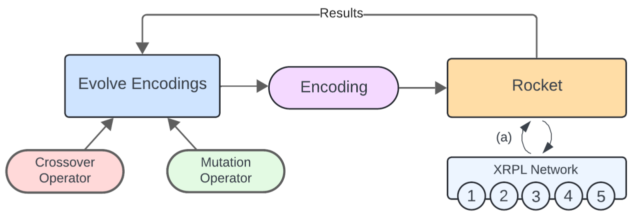

Figure 2: Overview of test case evolution.

Figure 2 provides an overview of how test cases are evaluated during the evolutionary process. An encoding, along with additional configuration parameters such as the number of nodes, is sent to Rocket. Rocket interacts with the XRPL network by intercepting messages and applying delays according to the given encoding (a). Once a test case has been executed, Rocket returns the results, which are then used to compute the fitness of the corresponding individual. After evaluating all individuals in the population, crossover and mutation operators are applied to generate offspring. The combined population of parents and offspring is then passed through NSGA-II selection to retain the best individuals to form a new generation. This process is repeated until the final generation has been evaluated.

3.4 Detecting Consensus Violations

To detect liveness and safety violations, we analyze the four BFT consensus properties (mentioned in Section 2.2) using the methodology presented in [4]:

- **Termination:** A violation is recorded if a consensus round fails to complete within a bounded time frame; specifically, if the time to reach agreement on a new ledger exceeds a predefined timeout of 65 seconds.
- **Validity:** A violation is recorded if a validator agrees on a transaction set that includes transactions not proposed by any validator, validates a ledger that was never constructed by any other validator, or switches to a ledger chain unsupported by the network.
- **Integrity:** A violation is recorded if a validator declares consensus more than once in the same round or sends multiple validation messages for the same ledger sequence.
- **Agreement:** A violation is recorded if two validators declare consensus on different transaction sets or validate different ledgers.

4 Evaluating Genetic Operators

We combine our methodology with a controlled experimental setup (Section 4.1) to evaluate how genetic operator selection influences delay-based evolutionary testing of the XRPL consensus algorithm. To structure this evaluation, we define a research question (Section 4.2) that guides our analysis of how genetic operator selection affects testing performance.

Table 2: Evolutionary configurations to compare.

Configuration Name	Description
The Fantastic (baseline)	Acts as a baseline configuration which completely randomizes every generation.
The Thing	Favors exploitation through SBX crossover ($\eta = 3$), but exploration through Gaussian mutation ($\mu = 0, \sigma = 40$).
The Torch	Favors exploration through Blend-α crossover ($\alpha = 0.7$) and Gaussian mutation ($\mu = 0, \sigma = 40$).
The Invisible	Favors exploitation through SBX crossover ($\eta = 3$) and localized mutation via polynomial mutation ($\eta = 20$).

4.1 Experimental setup

In our experiments[2], we test two versions of XRPL. The first is the official release version 2.4.0[3], which we refer to as version A. The second is a modified, seeded version of 2.4.0[4], in which the consensus threshold is lowered from 80% to 40%; we refer to this variant as version B. This reduction makes the network more prone to consensus violations, such as halted progress (liveness violations) or validators reaching agreement on different ledgers (safety violations), thereby increasing the likelihood of exposing bugs during testing. This controlled increase in bug likelihood enables a more informative comparison of the effectiveness of different genetic operators.

For each XRPL version, we test four evolutionary configurations, including one baseline configuration, which are listed in Table 2. These configurations combine different genetic operators to examine how varying levels of exploration and exploitation affect bug discovery. The baseline configuration, *The Fantastic*, applies full randomization in every generation. *The Thing* emphasizes exploitation in crossover using SBX and exploration in mutation via Gaussian mutation. *The Torch* emphasizes exploration in both crossover and mutation by combining Blend-α crossover with Gaussian mutation. *The Invisible* focuses on exploitation through SBX crossover and localized mutation using polynomial mutation.

The network runs with delays until all nodes have closed the ledger with sequence number 10, then proceeds without delay until ledger 14 to allow recovery from the perturbations. Delays are also lifted earlier if any node closes a ledger with sequence number 12.

Inspired by prior work [4], we use a similar transaction pattern involving three XRP accounts. Account 1 starts with 100,000 XRP and attempts to overspend by concurrently submitting four transactions of 80,000 XRP: two to Account 2 and two to Account 3. Specifically, we submit the following transactions to four different validator nodes: $\text{Tx}_1 = (1, 1, 2, 80\,000)$, $\text{Tx}_2 = (2, 1, 3, 80\,000)$, $\text{Tx}_3 = (3, 1, 2, 80\,000)$, and $\text{Tx}_4 = (4, 1, 3, 80\,000)$, where each tuple represents (transaction ID, source account, destination account, amount). This overspending pattern is designed to create race conditions, as the transactions compete to be included in the ledger despite the source account lacking sufficient funds to cover all of them. In the presence of network delays, such conditions can potentially lead to inconsistent validation outcomes across nodes, resulting in ledger divergence or other forms of consensus failure. Accounts are created after ledger sequence 2 is validated (ensuring proper network setup), and the transactions are dispatched 2000 ms later.

To analyze a test case, we extended the logging functionality in Rocket to capture the information necessary to verify consensus properties. Liveness failures are detected using built-in functionality provided by Rocket. To detect safety violations, we enhanced Rocket to poll the validated ledgers from the validator nodes after the test case has executed. This allows us to verify that no two nodes have declared consensus on different ledgers.

Tests run on a remote server (2× AMD EPYC 7H12, 128 cores, 256 threads, 256GB RAM). With an average runtime of 2.73 minutes per test, the full set totals roughly 7.58 days. Each test case is executed within an isolated Docker[5] container, ensuring clean separation between runs and preventing any cross-test interference.

4.2 Evaluation Criteria

To evaluate the effectiveness of different genetic operators in a delay-based evolutionary approach in testing the XRPL consensus algorithm, we will answer the following research question:

RQ *How does the selection of genetic operators affect the performance of guided randomized testing in an evolutionary approach?*

This research question serves multiple purposes. First, it allows us to compare the effectiveness of evolutionary testing against unguided randomized testing, providing insight into whether guided search strategies offer a measurable advantage. Second, it enables a detailed assessment of how the choice of genetic operators influences search dynamics and bug-discovery effectiveness. Finally, it helps us evaluate the overall capability of our testing approach to uncover meaningful bugs in the XRPL consensus algorithm.

To answer this question, we compare the performance of the four evolutionary configurations (Table 2), including the unguided randomized baseline, in terms of how many consensus violations they uncover and how quickly they do so. This allows us to assess whether the use of guidance through genetic operators leads to improvements in fault discovery compared to unguided search.

We also examine how the evolutionary configurations differ in their search behavior over time. Specifically, we analyze how each configuration balances exploration and exploitation, and how this influences the discovery of fault-revealing test cases in XRPL's complex input space. This helps us understand not only which configurations perform well, but why they do so.

Finally, we assess the broader effectiveness of the evolutionary testing method as a whole. Our goal is to determine whether the method provides a reliable and practical way to uncover consensus failures under our experimental conditions, and how it compares to findings from prior work on concurrency testing.

[2]The implementation of our experimental setup is available at https://github.com/amousavigourabi/rocket/tree/btgg

[3]https://github.com/XRPLF/rippled/releases/tag/2.4.0

[4]https://github.com/amousavigourabi/bug-seeded-rippled/tree/seeded-2.4.0-fully-lowered-threshold

[5]https://www.docker.com/

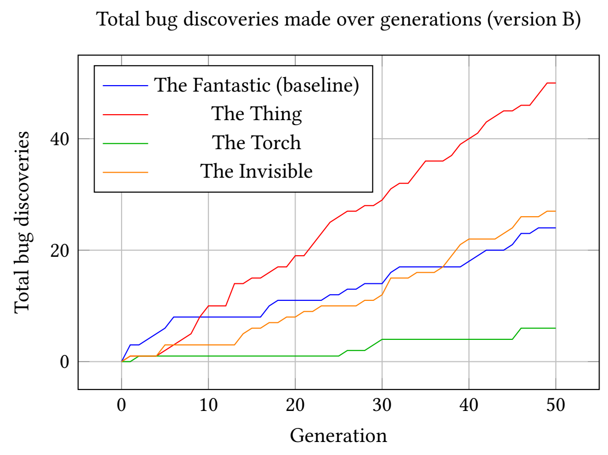

Figure 3: The trajectory of bug discoveries in XRPL version B.

5 Results

We now present the results that address our research question, covering baseline comparisons, the impact of operator choice, and the overall effectiveness of our evolutionary testing approach. Notably, no bugs were discovered in the unseeded XRPL version throughout the course of our experiments. As a result, our analysis primarily focuses on the bug-seeded version of XRPL, where the evolutionary process consistently uncovered consensus violations across multiple configurations.

Figure 3 shows the trajectory of the total amount of bug discoveries across generations in the bug-seeded XRPL version (version B) per configuration. A clear divergence in total discoveries emerges early on, highlighting performance differences between configurations. Among them, *The Thing* (SBX & Gaussian) consistently outperforms the rest, maintaining a strong linear discovery rate throughout the evolutionary process and ending with the highest overall count. *The Fantastic* (our baseline) and *The Invisible* (SBX & Polynomial) follow similar trajectories, showing moderate but sustained growth over generations. In contrast, *The Torch* (Blend-α & Gaussian) exhibits limited effectiveness, flattening out early and contributing few additional discoveries beyond generation 25.

These trends suggest that certain configurations, particularly *The Thing*, achieve a more effective balance between exploration and exploitation. This balance appears to support sustained bug discovery performance.

The Invisible eventually surpasses the baseline and appears to rise more rapidly over time. This behavior is presumably due to its exploitation-heavy design: once high-performing individuals are identified, the algorithm generates offspring that closely resemble them. This increases the likelihood of repeatedly triggering similar violation-prone executions, resulting in an accelerating discovery rate.

In contrast, *The Torch* heavily favors exploration and performs worse than the baseline. Its tendency to generate offspring that deviate significantly from high-performing individuals hinders the retention of beneficial traits. This behavior appears to cause the search to drift into unproductive areas of the input space, resulting in prolonged intervals of poor performance before discovering a new local optimum.

Table 3: Performance of configurations on XRPL versions A and B.

Configuration	**Total**		L_{20}		D_5		$\overline{f_t}$	
	A	B	A	B	A	B	A	B
The Fantastic (baseline)	0	24	0	10	–	4	4.27	5.06
The Thing (SBX & Gaussian)	0	50	0	22	–	8	4.26	5.32
The Torch (Blend-α & Gaussian)	0	6	0	3	–	46	4.35	4.20
The Invisible (SBX & Polynomial)	0	27	0	16	–	14	4.15	5.70

Table 3 presents the evaluation metrics for all configurations in XRPL versions A and B, representing the unseeded and seeded variants, respectively. The *Total* column shows the overall number of consensus violations discovered during the experiments. L_{20} captures the number of violations found in the last 20 generations, providing insight into long-term performance. D_5 indicates the generation at which each configuration reached its fifth discovery, serving as an indicator of early traction. Finally, $\overline{f_t}$ represents the average validation time per ledger across all evaluated individuals.

Looking at the totals for version B, we reach the same conclusion: *The Thing* yields the highest overall number of discoveries, confirming its superior effectiveness. *The Fantastic* and *The Invisible* show similar performance on this metric. However, L_{20} confirms our earlier observation: while *The Invisible* exhibits low early traction, it consistently outperforms the baseline in later generations, achieving a higher bug discovery rate over time. This is further supported by the D_5 metric, where *The Invisible* takes remarkably more generations than *The Fantastic* to reach its first five discoveries.

As also seen in Figure 3, The Fantastic demonstrates a notably consistent discovery rate throughout the evolutionary process. Out of 50 generations, it uncovered 24 bugs in total, with 10 of those found in the final 20 generations. This reflects a relatively steady rate of approximately one bug every two generations. This pattern contrasts with The Torch and The Invisible, which exhibit more fluctuation in their discovery rates over time. One notable exception is The Thing, which also follows a largely linear trajectory, but at a higher average rate of discovery.

The D_5 metric reveals that the baseline configuration (*The Fantastic*) reached five bug discoveries significantly earlier than the other configurations. This can be attributed to its fully randomized search strategy, which may have produced favorable, violation-prone executions early on. In contrast, the other configurations required more time to reach this milestone. Since their performance depends on gradually refining the population, a poor initial population can delay early progress. Suboptimal parents are likely to produce similarly ineffective offspring until stronger individuals emerge through selection and variation. In some cases, the initial conditions are so poor that even highly explorative configurations struggle to recover, as clearly exemplified by *The Torch*.

For $\overline{f_t}$, we observe no meaningful differences between configurations in the unseeded XRPL version A. This is expected, as no bugs were discovered in that version, suggesting the XRPL consensus algorithm is strongly resilient to delay-based perturbations.

In contrast, version B reveals clearer distinctions on $\overline{f_t}$. *The Torch* shows a notably low $\overline{f_t}$, comparable to values from version A, which aligns with its poor bug discovery performance. Meanwhile, *The Thing* and *The Invisible* both show higher values than the baseline, consistent with their superior effectiveness. *The Invisible* exhibits the highest $\overline{f_t}$ overall, potentially due to its exploitation-heavy behavior. Once a few high-f_t individuals are discovered, there is a high likelihood that subsequent offspring will retain similar performance, thereby sustaining elevated $\overline{f_t}$ values even when no additional bug discoveries are made.

6 Discussion

Our results highlight the impact of genetic operator selection on the effectiveness of evolutionary testing in uncovering consensus violations in XRPL. By varying the balance between exploration and exploitation through different operator configurations, we observed notable differences in bug detection performance.

The best-performing configuration favored exploitation in the crossover operator, effectively recombining promising individuals, while maintaining exploration through mutation, introducing occasional diversity. This combination created a superior balance, enabling the search to refine effective test inputs while still producing diverse behaviors. In contrast, configurations that lean too heavily towards exploration or exploitation performed worse. *The Torch*, the purely exploratory configuration, often failed to build on previously effective test cases, limiting its ability to refine promising behaviors. In contrast, *The Invisible*, which is heavily exploitative, tended to converge slowly toward a few high-performing individuals, reducing input diversity and missing other fault-inducing scenarios. This pattern of behavior aligns with expectations from prior research on the trade-off between exploration and exploitation in evolutionary algorithms [32].

These observations suggest that the XRPL input space contains clusters of bug-revealing inputs that are locally dense but globally sparse. In other words, once a promising area is discovered, local refinement (exploitation) is beneficial, but, occasional jumps (exploration) are necessary to escape local optima and discover new fault-prone regions.

As expected, our results also show that unguided randomized testing remains a strong baseline, and is not easily outperformed by guided evolutionary strategies. Because random testing generates completely unstructured inputs, it naturally produces highly diverse test cases, which can be more beneficial in exposing faults than guiding the search toward a specific optimum. This diversity allows random testing to cover unexpected scenarios that more targeted approaches might overlook.

When examining the average validation time, $\overline{f_t}$, we observe a clear correlation between higher $\overline{f_t}$ and the number of bugs found. This supports the idea that a long execution time is a meaningful indicator for bug detection effectiveness, as suggested in prior work [4].

We evaluated our approach using a bug-seeded version of XRPL v2.4.0, in which the same consensus bug was manually injected as in prior work using XRPL v1.7.2 [4]. That work reported a bug detection success rate of approximately 70% when evolutionary testing was run for an average of 18 generations. By comparing the first 18 generations of our approach under highly similar setup parameters, we establish a baseline for evaluating the efficiency of our method. This suggests that our testing strategy performs comparably in terms of effectiveness, while offering indications of improved efficiency in bug detection relative to prior work.

7 Threats to Validity

We acknowledge three threats to the validity of our results. While we designed our experiments to reflect realistic conditions and applied mitigation strategies where feasible, certain limitations remain due to the nature of the system under test and the testing methodology.

First, XRPL itself is inherently nondeterministic. As a result, even running the exact same test case multiple times can lead to different outcomes. For instance, transactions included in a ledger may vary across runs due to timing differences in message propagation and processing. These variations could not be controlled or eliminated, as our approach treats the system as a black box. We run a full live network and inject network faults without modifying or interfering with XRPL's internal logic. This means we have control over when messages arrive, but not over the exact timing or order in which nodes internally process and send those messages. However, we deliberately adopted this setup to reflect conditions as close as possible to real-world deployments, accepting the inherent nondeterminism as part of the testing environment. To partially mitigate this variability, we executed each test case twice when determining fitness, which provided a more stable and representative estimate of test effectiveness.

Second, we observed occasional false positives in liveness violations due to threading issues in the Rocket framework, which occur when the machine cannot handle the number of concurrent threads required for all node transmissions in the XRPL network. Under heavy load, some messages get queued unexpectedly, making it seem like nodes have stalled when they are only delayed. To deal with this, we manually inspected all reported liveness violations. In each case, the issue was either a false positive caused by message queuing or a real liveness failure that occurred after an earlier safety failure, where two nodes validated different ledgers. For the seeded bug experiments, we therefore only checked for this specific condition, since it is the concrete fault that the injected bug is expected to trigger.

Third, evolutionary testing is inherently stochastic, meaning that repeated runs may produce different sequences of test inputs and slightly different bug discovery trajectories. While the overall conclusions are generally stable, individual runs can vary in detail. To mitigate this, we repeated each experiment three times and verified that the key measurements and overall bug discovery patterns remained consistent across runs. We also used a fixed random seed for each run to improve reproducibility. However, due to nondeterminism in the XRPL implementation itself, this had limited effect on ensuring identical outcomes. Nevertheless, the consistency of results across runs supports the reliability of our conclusions.

8 Responsible Research

Throughout this research, we have followed and will continue to follow ethical standards to ensure the integrity, reproducibility, and responsible publication and reporting of our work.

First, we made reproducibility a core consideration in the design and execution of our experiments. The complete setup used in our evaluation is described in detail, and we used the exact same environment for all experimental runs. To support repeatability, we used fixed random seeds to initialize each run. However, as discussed earlier, exact repeatability remains challenging due to the inherently nondeterministic nature of XRPL, where variations in timing and message delivery can lead to different outcomes across executions. Our results and the software used in our experiments are available at Zenodo [33].

Second, in the event that our methodology reveals a potential real-world consensus violation in XRPL, we commit to the following coordinated and responsible disclosure practices. Any such discovery will be privately reported to the Ripple development team to give them the opportunity to assess and address the issue before any public disclosure is considered. We recognize that consensus failures in XRPL could have significant financial implications for its users and the broader cryptocurrency ecosystem, and as such, premature disclosure could pose serious risks. Our approach prioritizes user safety, network stability, and the integrity of the XRPL platform.

9 Conclusion and Future Work

In this paper, we evaluated the impact of different genetic operator configurations in delay-based concurrency testing of the XRPL consensus algorithm using an evolutionary approach. Our experiments explored configurations with varying balances of exploration and exploitation, with the goal of effectively discovering bugs. We found that configurations combining exploitation in crossover with exploration in mutation perform particularly well. In contrast, configurations that lean too heavily toward either exploration or exploitation tend to perform worse. Excessive exploitation leads to low input diversity and struggles to discover new fault-revealing areas, while excessive exploration fails to build on promising inputs. The strong performance of unguided random testing highlights that maintaining diversity and unpredictability in test generation continues to play an important role in effective fault discovery, especially in complex and high-dimensional spaces where structural assumptions may not hold.

From these findings, we can conclude that the XRPL input space for delay-based representations is likely characterized by locally dense clusters of fault-revealing inputs, separated by large, sparse regions. This suggests that once a promising region is found, focused refinement through exploitation is effective, but occasional exploratory jumps remain essential for discovering new clusters of faults.

For future work, a natural extension would be to explore a broader range of genetic operators, particularly those designed for real-valued input representations. Further improvements may also be achieved by tuning key hyperparameters such as population size, mutation rate, and selection strategies, as well as operator-specific parameters that influence the behavior of crossover and mutation. Systematic experimentation with these hyperparameters could lead to more effective search dynamics and improved fault discovery. Beyond this, our methodology could be applied to detect more complex or less subtle seeded bugs, allowing for a deeper assessment of its fault detection capabilities.

Another promising research direction is the incorporation of self-learning techniques. These methods have shown success in testing other distributed systems and could enhance the adaptability and precision of testing strategies for XRPL.

Finally, this delay-based concurrency testing approach could also be applied to other blockchain platforms. Doing so may provide valuable comparative insights and help uncover the structure and nature of input spaces in different consensus systems.

References

[1] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (1982), pp. 382–401. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176.

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

[3] Levin N. Winter et al. "Randomized Testing of Byzantine Fault Tolerant Algorithms". In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 757–788. DOI: 10.1145/3586053. URL: https://doi.org/10.1145/3586053.

[4] Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. "Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm". In: *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023.* IEEE, 2023, pp. 36–47. DOI: 10.1109/ICSE-SEIP58684.2023.00009. URL: https://doi.org/10.1109/ICSE-SEIP58684.2023.00009.

[5] Wishaal Kanhai et al. "Rocket: A System-Level Fuzz-Testing Framework for the XRPL Consensus Algorithm". In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST).* 2025, pp. 737–741. DOI: 10.1109/ICST62969.2025.10988979.

[6] Marios Touloupou et al. "Benchmarking Blockchains: The case of XRP Ledger and Beyond". In: *55th Hawaii International Conference on System Sciences, HICSS 2022, Virtual Event / Maui, Hawaii, USA, January 4-7, 2022.* ScholarSpace, 2022, pp. 1–8. URL: http://hdl.handle.net/10125/80070.

[7] David H. Wolpert and William G. Macready. "No free lunch theorems for optimization". In: *IEEE Trans. Evol. Comput.* 1.1 (1997), pp. 67–82. DOI: 10.1109/4235.585893. URL: https://doi.org/10.1109/4235.585893.

[8] Orestis Alpos et al. "Asymmetric distributed trust". In: *Distributed Comput.* 37.3 (2024), pp. 247–277. DOI: 10.1007/S00446-024-00469-1. URL: https://doi.org/10.1007/s00446-024-00469-1.

[9] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3. URL: https://doi.org/10.1007/978-3-642-15260-3.

[10] Paul Thomson, Alastair F. Donaldson, and Adam Betts. "Concurrency Testing Using Controlled Schedulers: An Empirical Study". In: *ACM Trans. Parallel Comput.* 2.4 (2016), 23:1–23:37. DOI: 10.1145/2858651. URL: https://doi.org/10.1145/2858651.

[11] Kyle Kingsbury. *Jepsen.* http://jepsen.io/. 2022.

[12] Fuchen Ma et al. "LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols". In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023.* The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/loki-state-aware-fuzzing-framework-for-the-implementation-of-blockchain-consensus-protocols/.

[13] Sebastian Burckhardt et al. "A randomized scheduler with probabilistic guarantees of finding bugs". In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010.* Ed. by James C. Hoe and Vikram S. Adve. ACM, 2010, pp. 167–178. DOI: 10.1145/1736020.1736040. URL: https://doi.org/10.1145/1736020.1736040.

[14] Xinhao Yuan and Junfeng Yang. "Effective Concurrency Testing for Distributed Systems". In: *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020.* Ed. by James R. Larus, Luis Ceze, and Karin Strauss. ACM, 2020, pp. 1141–1156. DOI: 10.1145/3373376.3378484. URL: https://doi.org/10.1145/3373376.3378484.

[15] Cezara Dragoi et al. "Testing consensus implementations using communication closure". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 210:1–210:29. DOI: 10.1145/3428278. URL: https://doi.org/10.1145/3428278.

[16] Suvam Mukherjee et al. "Learning-based controlled concurrency testing". In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 230:1–230:31. DOI: 10.1145/3428298. URL: https://doi.org/10.1145/3428298.

[17] Phil McMinn. "Search-Based Software Testing: Past, Present and Future". In: *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Berlin, Germany, 21-25 March, 2011, Workshop Proceedings.* IEEE Computer Society, 2011, pp. 153–163. DOI: 10.1109/ICSTW.2011.100. URL: https://doi.org/10.1109/ICSTW.2011.100.

[18] Mohammad Moein Almasi et al. "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application". In: *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017.* IEEE Computer Society, 2017, pp. 263–272. DOI: 10.1109/ICSE-SEIP.2017.27. URL: https://doi.org/10.1109/ICSE-SEIP.2017.27.

[19] Pouria Derakhshanfar et al. "Generating Class-Level Integration Tests Using Call Site Information". In: *IEEE Trans. Software Eng.* 49.4 (2023), pp. 2069–2087. DOI: 10.1109/TSE.2022.3209625. URL: https://doi.org/10.1109/TSE.2022.3209625.

[20] Andrea Arcuri. "EvoMaster: Evolutionary Multi-context Automated System Test Generation". In: *CoRR* abs/1901.04472 (2019). arXiv: 1901.04472. URL: http://arxiv.org/abs/1901.04472.

[21] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3. URL: https://doi.org/10.1007/978-3-642-15260-3.

[22] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Commun. ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

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

[24] Tianshi Chen et al. "A large population size can be unhelpful in evolutionary algorithms". In: *Theor. Comput. Sci.* 436 (2012), pp. 54–70. DOI: 10.1016/J.TCS.2011.02.016. URL: https://doi.org/10.1016/j.tcs.2011.02.016.

[25] Tinkle Chugh et al. "A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms". In: *Soft Comput.* 23.9 (2019), pp. 3137–3166. DOI: 10.1007/S00500-017-2965-0. URL: https://doi.org/10.1007/s00500-017-2965-0.

[26] Kalyanmoy Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE Trans. Evol. Comput.* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017. URL: https://doi.org/10.1109/4235.996017.

[27] Kalyanmoy Deb and Ram Bhushan Agrawal. "Simulated Binary Crossover for Continuous Search Space". In: *Complex Syst.* 9.2 (1995). URL: http://www.complex-systems.com/abstracts/v09%5C_i02%5C_a02.html.

[28] Kalyanmoy Deb. "An efficient constraint handling method for genetic algorithms". In: *Computer Methods in Applied Mechanics and Engineering* 186.2 (2000), pp. 311–338. ISSN: 0045-7825. DOI: 10.1016/S0045-7825(99)00389-8. URL: https://doi.org/10.1016/S0045-7825(99)00389-8.

[29] Larry J. Eshelman and J. David Schaffer. "Real-Coded Genetic Algorithms and Interval-Schemata". In: *Proceedings of the Second Workshop on Foundations of Genetic Algorithms. Vail, Colorado, USA, July 26-29 1992.* Ed. by L. Darrell Whitley. Morgan Kaufmann, 1992, pp. 187–202. DOI: 10.1016/B978-0-08-094832-4.50018-0. URL: https://doi.org/10.1016/b978-0-08-094832-4.50018-0.

[30] Sofiane Achiche, Luc Baron, and Marek Balazinski. "Scheduling exploration/exploitation levels in genetically-generated fuzzy knowledge bases". In: July 2004, 401–406 Vol.1. ISBN: 0-7803-8376-1. DOI: 10.1109/NAFIPS.2004.1336316. URL: https://doi.org/10.1109/NAFIPS.2004.1336316.

[31] Kalyanmoy Deb and Debayan Deb. "Analysing mutation schemes for real-parameter genetic algorithms". In: *Int. J. Artif. Intell. Soft Comput.* 4.1 (2014), pp. 1–28. DOI: 10.1504/IJAISC.2014.059280. URL: https://doi.org/10.1504/IJAISC.2014.059280.

[32] Matej Crepinsek, Shih-Hsi Liu, and Marjan Mernik. "Exploration and exploitation in evolutionary algorithms: A survey". In: *ACM Comput. Surv.* 45.3 (2013), 35:1–35:33. DOI: 10.1145/2480741.2480752. URL: https://doi.org/10.1145/2480741.2480752.

[33] Wishaal Kanhai. *Evaluation Results and Software Tools for Genetic Operators in Delay-Based Testing of the XRPL Consensus Algorithm.* June 2025. DOI: 10.5281/zenodo.15700437. URL: https://doi.org/10.5281/zenodo.15700437.