Groot: Impact of Evolutionary Operators in XRPL Testing using Priority-Based Event Representation

Bryan Wassenaar[1]

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

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

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

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

Groot: Impact of Evolutionary Operators in XRPL Testing using Priority-Based Event Representation

Bryan Wassenaar

Delft University of Technology
Delft, The Netherlands

Abstract

The decentralized nature of blockchain systems makes them prone to concurrency bugs, which are difficult to detect. There exist testing techniques to find these bugs, such as systematic exploration of the solution space, but these techniques are difficult to scale. Evolutionary algorithms have been proposed as an effective solution to find these bugs.

In this research, we aim to discover the influence of evolutionary operators in the bug detection performance of evolutionary algorithms. We test this on the XRP Ledger Consensus Protocol (XRP LCP) using priority-based event representation. We present Groot, an evolutionary algorithm that is implemented using a modified version of the Rocket framework. We experimented with two combinations of operators: the Simulated Binary Crossover (SBX) operator with the Gaussian mutation operator and the Laplace Crossover (LX) operator with the Makinen, Periaux and Toivanen Mutation (MPTM) operator. We evaluated these setups using effectiveness and efficiency and compared them to a random baseline. We used a bug-seeded version of the XRP LCP to run the experiments of these setups.

We discovered that all setups are capable of detecting bugs in the XRP LCP. The results indicate that the effectiveness and efficiency is not influenced by the choice of these operators in a significant way. We discuss that possible reasons for these discoveries include noise in the fitness function, event representation limitations and configuration choices that may have contributed to these results.

Reference Format:
Bryan Wassenaar. 2025. Groot: Impact of Evolutionary Operators in XRPL Testing using Priority-Based Event Representation. In . TU Delft, Delft, NL, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

With a market capitalization of 2.6 trillion USD in 2021, crypto-assets can become a threat to global financial stability according to the Financial Stability Board [1]. Unlike traditional banking systems, cryptocurrencies are decentralized systems which processes transactions without a central authority which are most commonly build on blockchains.

One important part of the blockchain is the consensus protocol. This protocol describes how nodes should communicate to reach agreement within the network. The consensus protocol for XRP is the focus of this research. This protocol is theoretically secure as proposed in the paper from Schwartz [19]. However, implementation of these systems is not perfect and concurrency bugs can cause faults such as double spending or network forks. These only occur in specific message ordering or timings and are difficult to detect [11]. This means that the consensus protocol should be tested thoroughly to avoid such bugs happening in production.

One way to test such a system is by using an evolutionary algorithm, which was proven effective in recent research [21]. These algorithms search for test cases in which faults or bugs occur, by using a similar method to biological evolution. The evolutionary algorithms are split into selection and reproduction. The focus of this research is the reproduction part which is split in crossover and mutation operators. The crossover operator takes one or more parent cases and creates one or more child cases by combining the parents in some mathematical way. The mutation operator changes one or more values within a child case to create more diversity.

In the before mentioned research [21], there were some gaps that could be explored more deeply. First of all, the previous research explores only two fitness function, time-fitness and proposal-fitness. Secondly, it uses only one combination of crossover and mutation operators, the simulated binary crossover operator [5] with the gaussian mutation operator [6]. More operator combinations can be researched to discover the influence of these operators on the performance of the evolutionary algorithm. Last of all, even though the research described two event representations, only the delay-based method is used in the evolutionary algorithm. The last gap was discussed in other work from van Meerten [20].

This study researches the gap of the lack of evolutionary operators and the priority-based event representation. We do the experiments using the rocket framework [10]. This framework is designed to test the XRPL consensus protocol by intercepting messages from nodes and delaying, dropping or manipulating them. The operators that are evaluated are Simulated Binary Crossover (SBX) with Gaussian mutation and Laplace crossover (LX) with MPT mutation. This work evaluates these operators with the time fitness function and priority-based event representation.

Our results show that the effectiveness of the algorithms does not differ in significant amounts between the baseline and evolutionary algorithms. It also shows that efficiency is possibly influenced by the choice of evolutionary operators.

Our contributions with this research are the following:

- An evolutionary algorithm called Groot for testing the XRPL protocol.

Bachelor Thesis'25, Delft, NL

https://doi.org/XXXXXXX.XXXXXXX

- A Reproduction Package to replicate this research.
- A comparison between two combinations of operators against a random baseline.

2 Background

In the background we will go over the workings of the XRP Ledger protocol and evolutionary algorithms. We will also discuss work related to this research such as other ways of testing distributed systems and different evolutionary operators.

2.1 XRP Ledger Consensus Protocol

In blockchains there is no central authority which validates transactions, unlike in a traditional banking systems. Blockchain systems store information about accounts and transactions on a block. These blocks are stored by every node in the network and chained together to create the ledger. The blocks are also immutable so that the contents cannot be changed after a ledger has been validated. The nodes have to agree which transactions are committed to the ledger and which are declined, such that there is only one version of the ledger in the whole network.

The XRP LCP is such a system and works on the basis of consensus. Transactions can be submitted to any of the validator nodes in the network. The network validates transactions by having the nodes agree on a set of transactions and commit them to a ledger. XRP LCP is byzantine fault-tolerant, meaning that it can cope with malicious participants in the network. The participation in the network is also open, each node defines a set of trusted participants in a Unique Node List (UNL). The node runs the protocol using the votes from these nodes. As long as 80% of the nodes are non-byzantine, the protocol guarantees correctness [19].

The protocol consists of synchronized consensus rounds in which transactions get added, proposed for the current ledger and the ledger gets validated and added to the chain. A consensus round consists of three stages: open, establish and validate. Transactions can be submitted to the nodes at any time, but they will likely be included in the next round.

2.1.1 Open. In the open phase, the nodes distribute submitted transactions to the other nodes in their UNL. They collect all transactions to be included in the next ledger to prepare for the establish phase. The open phases takes about half the duration of the previous consensus round.

2.1.2 Establish. During the establish phase the nodes will try to reach consensus on the set of transactions to include in the next ledger. They do this by iteratively proposing sets of transactions until a consensus of at least 80% is reached. Of course not every node has the same transaction set because of network faults and timings. Transactions that are not included in the nodes own proposals but are in other node's proposals or transactions that are in the node's own proposal, but not in the proposals of other nodes are called disputed transactions. Every node makes a list of these disputed transactions. During the phase, nodes change their proposals to include transactions that most other nodes in the UNL also include and by removing disputed transactions that are not supported by the other nodes. Transactions become more easily disputed when the round takes longer compared to the previous round. At the start transactions are marked as disputed if less than 50% of the UNL nodes propose it. This percentage increases to 65%, 70% and 95% as the round progresses. When the 80% agreement is reached, a node will declare consensus and move on to the validation phase. If no agreement can be reached before a predetermined time, then the nodes will go back to the open phase.

2.1.3 Validate. The final phase validates if all nodes agreed on the same ledger. The nodes do this by computing a ledger hash from the agreed transaction set and sharing this with the other nodes. Once a node has collected validations with the same hash from more than 80% of the nodes in its UNL, it will assume the ledger is fully validated. It will then apply all transactions to the ledger, which will become immutable.

2.1.4 Correctness properties. The specification of correctness that we used in this research is the following [3]:

- **Termination**: Every correct process eventually decides some value.
- **Validity**: If a process decides some value, then that value was proposed by some process.
- **Integrity**: No process decides twice.
- **Agreement**: No two correct processes decide differently.

In the case of the XRP LCP the value to decide upon is which ledger to validate and a process is a node. The two most important properties for this research are the agreement and termination properties. An example of an agreement violation would be when two nodes validate two different ledgers, causing a fork in the network. An example of a termination violation would be if the network gets stuck and stops processing any transactions.

2.2 Evolutionary algorithm

Evolutionary algorithms work on the principle of biological evolution. It is specifically useful for search spaces which are complex. The algorithm works with populations. A population is a group of test cases for one round, also called a generation. In our research we also call an individual test case an encoding. An evolutionary algorithm always starts with an initial population. This initial population is often randomly generated. Each population performs four stages. First all test cases in the population are tested. After this some selection takes place to determine the mating pool. After selection crossover and mutation operators are applied to create the new population for the next generation from the mating pool. After this the new population gets run through the same cycle. This repeats until a pre-determined generation or some result has been reached.

2.2.1 Selection. Within the selection it is determined from the results of the test cases which test case performed best. This is decided by a fitness function, which calculates a value from the results for each test case, and a selection method.

For example tournament selection, which works by taking a small random sample of candidates from the population which will compete in a tournament. The candidate with the highest fitness is placed in the mating pool. This repeats until the desired size of the mating pool is reached.

2.2.2 Crossover operator. The crossover operator is responsible for combining test cases in the mating pool to create new test cases. The goal is a combination of exploitation and exploration. The crossover operators in our research work by removing two parents from the mating pool and combining them using the chosen crossover operator to create two children. These two children get added to the mutation pool, which will be used by the mutation operator.

2.2.3 Mutation operator. The mutation operator is the last step in creating a new generation. Its goal is to create more exploration within the search space. It does this by changing values within the test cases randomly to some degree using the mutation operator. It also limits the ranges of the values so that it does not go out of bounds.

2.3 Related Work

2.3.1 Testing distributed systems. There are many other ways to test a distributed system as well. First of all you can systematically explore possible executions of the system, which will eventually test all possible executions but is very time consuming [9], [8], [18]. You can also use randomized concurrency testing, which is faster than systematically exploring all options but can also miss certain executions because of the randomization [4] [12]. A third option is probabilistic concurrency testing, which uses controlled randomization with formal guarantees which aim to find bugs more efficiently than standard randomized testing, however these formal guarantees are difficult to reach in practice [2] [16]. Some other notable options include Reinforcement learning, which can be used to learn effective strategies based on previous executions [14], and ByzzFuzz, which is effective at finding implementation bugs in blockchain systems by applying small mutations to message contents [23]. As a final option evolutionary algorithms were also proven effective at finding bugs in the XRP Ledger Consensus Protocol (XRP LCP) [21].

2.3.2 Evolutionary operators. Evolutionary algorithms have been researched extensively and so a lot of operators exist for them. For example for crossover operators you have the simulated binary crossover (SBX) [5], Laplace crossover (LX) [7], dominance and co-dominance operators [17] or logistic crossover [15]. Also the following mutation operators are known in research: gaussian mutation [5], non-uniform mutation (NUM), Makinen, Periaux and Toivanen Mutation (MPTM) [13] and Power mutation (PM).

3 Groot

In this research, we compare the two combinations of operators to a random baseline. This is done with an experimental approach using a modified version of the Rocket framework and our own evolutionary algorithm called Groot. In this section we discuss the encoding, the workings of Groot including the operators and how we evaluate our setups.

3.1 Encoding

The behavior of the original Rocket framework was only able to use delays. For the priority-based event representation we modified Rocket to take a list of priorities, called the encoding. This encoding determines the order of message delivery for specific message types and nodes. Messages are intercepted and put in a queue for later delivery. A time interval depending on the queue size determines how often a message is delivered. The interval is calculated using algorithm 1 which was adjusted from van Meerten [21]. Messages with a lower priority value will be delivered before messages with a higher priority. Every time a message is delivered, the priority of all messages in the queue will be lowered to avoid messages getting delayed forever. Message delay is also capped at a maximum of one second so that delays are never responsible for faults, but instead the order of the messages is always the reason for faults.

Algorithm 1: Adjust Message Delivery Rate

if *inbox_size* > *target_inbox* × *overflow_factor* **then**
  $r \leftarrow \min(r \times \text{sensitivity\_ratio}, \text{max\_events})$;
else if *inbox_size* < *target_inbox* × *underflow_factor* **then**
  $r \leftarrow \max(r/\text{sensitivity\_ratio}, \text{max\_events}/6)$;
$\text{packets\_per\_sec} \leftarrow \max(\text{min\_packets\_per\_second}, \lfloor r \rfloor)$;

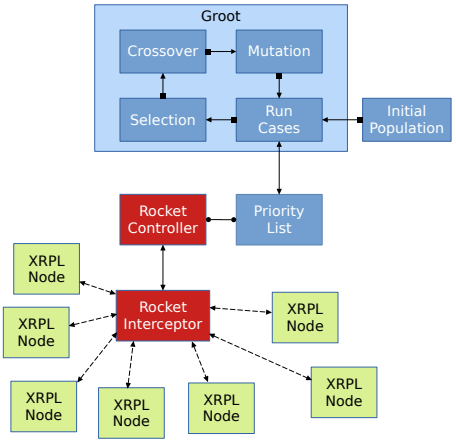

Figure 1: Experimental setup using the Rocket framework

3.2 The workings of Groot

Groot, our evolutionary algorithm, consists of several stages which can be seen in figure 1. First an initial population of random priority lists are created. Each of these encodings is run using the Rocket framework. The time that each encoding takes to validate a single ledger is measured and averaged to be used in selection, this is called time fitness. For time fitness we try to maximize the ledger validation time which can lead to more complex executions. crossover and mutation operators are applied to create a new population of priority lists. After this the new test cases can be run and the cycle repeats.

3.3 Random Baseline

The baseline setup is very simplistic. The same mechanism to create initial populations is used. However, instead of using selection and crossover with mutation operators, the baseline simply creates new populations by calling the initial population functionality again for

every generation. This does not take into account the probability that an encoding occurs twice.

3.4 Simulated Binary Crossover with Gaussian mutation

This setup was inspired by the research from van Meerten [21]. They also used the SBX and Gaussian operators which makes our research better comparable to theirs. This setup has a clear selection, crossover and mutation separation.

3.4.1 Selection. For selection we decided to use tournament selection, as this method is proven useful in evolutionary algorithms in other research such as from Naqvi [15]. In tournament selection, a small random sample is taken from the previous population to be candidates. The best sample from these candidates is selected and added to the mating pool. The samples are evaluated based on the before mentioned time fitness. This is repeated until the mating pool is the desired size.

3.4.2 SBX Crossover. After selection the simulated binary crossover operator is used to perform the crossover. Crossover takes place by selecting two parents randomly from the mating pool and applying the SBX operator to get two children. These children are added to the mutation pool while the parents are removed from the mating pool. This makes sure that every parent is used exactly once. This repeats until the mating pool is empty. The distribution value used for the operator is set to 3, to make our research better comparable to the research from van Meerten [21] who also used this value. It is also within the recommended range from the original paper [5].

3.4.3 Gaussian Mutation. Finally mutation takes place on the new mutation pool using the Gaussian mutation operator. Mutation is always performed on every encoding, but not on all single priorities. For every priority in the encoding the priority is mutated with a probability of 10%. If it is chosen for mutation then the Gaussian mutation operator is applied to that priority. After the mutation is done on the new population all values are restricted to be within the specified ranges.

3.5 Laplace crossover with MPT mutation

The last setup is inspired by papers such as from Naqvi [15]. Here they used Laplace and Makinen, Periaux and Toivanen Mutation as examples to compare their new operator with. The selection part for this setup is identical to the one from SBX and Gaussian mutation and will therefor be skipped in this section.

3.5.1 Laplace crossover. The Laplace crossover is used as an alternate distribution to the simulated binary crossover operator. It still uses the same method of choosing parents by selecting them from the mating pool and removing them after use so every parent is used exactly once. For the location and scaling variables we used 0 and 0.5 respectively as recommended by the original paper from Deep [7].

3.5.2 MPT Mutation. MPTM is a mutation operator which focuses on exploration early on and moves to exploitation in later generations. This operator was proposed by Mäkinen in 1999 [13] and has been used in other research since then. Just like in setup two, every priority inside an encoding is mutated with a probability of 10%, in which case the MPTM operator gets applied. We decided to use a scaling variable of 9 for a good balance between exploration and exploitation.

4 Study design

In our research, we divided the tests in three experimental setups. Experimental setup one is the random baseline without any operators. Setup two is the simulated binary crossover in combination with the Gaussian mutation operator, similarly to the paper from van Meerten [21]. The last setup is the Laplace crossover with MPT mutation, inspired by research from Naqvi [15]. The main research question we aim to answer is: How do different operators influence the ability of an evolutionary algorithm to detect bugs in implementations of the ripple consensus algorithm using priority-based event representation? We aim to answer this question with the following sub-questions:

- Can the evolutionary algorithms find any bugs in the implementation of the XRP Consensus Protocol or its variants?
- How is the effectiveness of bug detection performance influenced by the selected evolutionary operators?
- How is the efficiency of bug detection performance influenced by the selected evolutionary operators?

During this research, the Rocket framework was modified extensively. To accommodate this all code and docker images are available in a replication package on Zenodo [22]. All tests were run in docker on a physical server with one i5-11400F @2.60GHz processor and 16GB memory. Only one test was run at the same time to avoid overloading the docker daemon and all XRP nodes were configured as tiny nodes. All configurations discussed in the next sub sections are already configured in the provided docker images.

4.1 XRPL Network

For the runs of the XRPL network we decided to use seven nodes. We made this decision to create the possibility of agreement violations. The protocol has a minimum agreement of more than 80% of its peers [19], meaning that is less than seven nodes are used, all nodes would have to agree to reach consensus. When using seven nodes, it is possible that one node that does not agree with the others and the network still reaches consensus of more than 80%.

For the UNL of the nodes, it was decided to have almost identical UNL's. All nodes trust node 0,1,2,3 and 4, but node 5 is only trusted by nodes 0,1 and 2 whereas node 6 is only trusted by nodes 3 and 4. Nodes 5 and 6 only trust the nodes 0,1,2,3 and 4. By having almost identical UNL's the tests become less dependent on the UNL. We choose to not have full identic UNL's to create clearer possibilities for forks in the network, which in turn would become easier to detect.

The XRPL docker image used for testing is not the official XRPL image, but instead a bug seeded one where the agreement boundary is reduced to 40%. This makes it possible to be certain that our algorithm can find some bugs during testing. Due to the scope of the research it was not possible to also test our setups on the official XRPL image. The nodes are also configured as tiny nodes to make simulation tests possible.

4.2 Groot

Groot was run with a population size of 10 for 50 generations. These values were chosen to allow for enough generations to see a difference over time, but still keep within the scope of this research. It was decided to only use priorities for eight types of messages, as to not mess with the startup or synchronization of the nodes. Instead we only manipulate messages from transactions and ledgers. Since we have seven nodes in our XRPL network and we have eight types of messages, this means that we need encoding of length 294. Because we have encodings of length 294, we decided to use priorities ranging from 0 to 300. This way an absolute ordering of messages can be made with a high probability that small changes in priority lead to different orderings. From Groot, two test cases are run at the same time to speed up the testing process. If more tests were run in parallel problems with the docker daemon occurred causing containers to not be removed and thus leading to memory leaks.

4.3 Rocket

Modifications were made to the rocket framework to make it usable in this research. The interceptor was rewritten to allow multiple instances to run in parallel using docker and to allow running with custom XRPL images. The controller was extended to allow priorities to be used and to check violations with more accuracy. When testing an encoding, it is run twice in an iteration and the results from both iterations is averaged to compensate for concurrency variability. The reason we have only two iterations per encoding is because the sequential computation time of one test setup is 233 hours or around ten days. We were able to decrease this time with parallelization to around two days, but because of the amount of containers needed to be created, started, stopped and removed for each run, the docker daemon was not able to keep up with multiple setups at the same time. The averaged result is used in selection. During one iteration, a total of 14 ledgers are validated after which the iteration is ended. The first 10 ledgers are influenced by the priorities while the last four ledgers are run as normal to allow the network time to heal. During the iteration, three accounts are created with a balance of 80 XRP. Two seconds after the start of the iteration account one submits four transactions of 80 XRP to four different nodes. This can cause agreement violations and tries to commit a double spend. In the implementation every transaction has slightly different amounts to avoid the network detecting the transactions as one transaction.

4.4 Evaluation of the experimental setups

We evaluate the experimental setups on effectiveness and efficiency to discover the influence of the operators between the setups. Effectiveness is measured by the amount of generations which found a violation. Efficiency is measured based on the earliest generation that detected any violations and the number of violations identified within that generation. We check both agreement and termination violations. We mark a potential agreement violation when, at the end of an iteration, the final validated ledger hashes of all nodes do not match. We mark a potential termination violation when a ledger validation takes more than 65 seconds [21]. It is important to note that the ledger validation under normal procedures only takes a few seconds. We do not check for integrity violations, because the rocket framework does not incorporate data to determine this.

	Baseline	SBX-Gaussian	Laplace-MPTM
Failed Agreement	68	69	67
Failed Termination	0	0	0

Table 1: Total amount of runs having violations.

	Baseline	SBX-Gaussian	Laplace-MPTM
First Violation	G2	G1	G2
Failed Agreement	4	3	5
Failed Termination	0	0	0

Table 2: First generation that found a violation.

5 Results

The results section is split in two distinct parts. First we will take a look at the effectiveness of finding violations by comparing the amount of violations found for each setup. Second, we will compare the efficiency of the setups in finding the first violation.

5.1 Effectiveness in finding violations

Effectiveness of the setups is determined by the amount of violations that were found during the tests. From table 1 we can see that the amount of violations found does not differ a lot in between each setup. None of the setups was able to find termination failures. This is also seen from the fact that the maximum time a ledger validation took was only 9.74 seconds, which is lower than the 65 second limit for termination failures. The average of all ledger validation times was only 4.86 seconds. For agreement failures all setups were able to discuss a good amount of violations. From table 1 we can see that the baseline has 68 violations, SBX-Gaussian has 69 violations and Laplace-MPTM has 67 violations. There is no significant difference between the setups, as they only differ by 1 violation found. This means that all setups were as effective as each other. If we look at individual runs instead of populations, we can see that the Baseline has 915 correct runs without violations while the evolutionary approaches have 910 correct runs. This difference is again small, only 0.5%, which makes this difference not significant.

5.2 Efficiency in finding the first violation

Efficiency is determined as the amount of generations it took until the first violations was found and how many violations were found in that first generation. If we look at table 2 we can see that SBX-Gaussian was the first to find a violation in G1, however this was the initial generation. If we leave out G1 we see that the baseline and Laplace-MPTM were the first to find a violation in the second generation. Short after in generation three SBX-Gaussian was able to find its first violation. The difference between these generations is negligible, only one generation from 50.

We can also see that the amount of violations that the setups were able to find in its first generation are also quite close. The baseline found 4 agreement violations in its first generation, while

Laplace-MPTM found 5 agreement violations in generation 2. SBX-Gaussian was the slowest, finding only 3 agreement violations in generation 3. This means that with a maximum of 10 violations in one generation, we can see a difference between 10% and 20%.

6 Discussion

Within the discussion we will look at possible reasons that our results are close together for both effectiveness and efficiency. First we will discuss the reasons for effectiveness and secondly the reasons for efficiency.

6.1 Reasons for similar effectiveness

From the results we can see that effectiveness is not influenced by the choice of operators in evolutionary testing of the XRPL protocol. This could be because of the noise in the fitness function or the chosen event representation.

6.1.1 Noisy fitness function. It could be that the chosen configuration of two iterations in combination with time-fitness leads to too much variability within the fitness function. This creates less effectiveness, but this could be improved with more iterations. We were able to see from analyzing the logs that the variation between validation times could change with more than a few seconds between runs. This indicates enough variation that selection is influenced by this variation leading to almost random selection instead of selection based on the fitness values. Using more iterations would better average out the variation between rounds and possibly support better selection. This could in turn give better results for the evolutionary algorithms.

Another configuration possibility is the chosen fitness function. We used time-fitness as the fitness function, which worked well with delay-based event representation in the paper from van Meerten [21]. It could be that time-fitness does not work well with priority-based event representation because no extra delays are added during the test.

6.1.2 Event representation. It is also possible that priority-based event representation is not as effective at finding violations as delay-based event representation. This aligns with the research of van Meerten [21] [20] in which they discovered that delay-based event representation outperformed priority-based event representation using evolutionary algorithms. We have also seen that our research did not discover any termination violations. This is logical as priority based event representation does not cause large delays in its implementation, being capped to a delay of one second. Delay based event representation does implement larger delays getting closer to the termination threshold with every generation. In the paper from van Meerten [21] they found a termination bug which occurred because of the sum of the added delays to the messages. We have a maximum validation time of 9.74 seconds and would thus not reach this delay threshold to cause this bug.

6.2 Reasons for similar efficiency

We see that for efficiency the three setups also are close to each other. We do see that SBX-Gaussian is slightly worse than the baseline and Laplace-MPTM, being both one generation later and discovering less violations during the generation. Reasons for this could be the seeded bug being too easy to find or that not enough tests were run to take an average.

6.2.1 Seeded bug too easy to find. It could be that agreement threshold of 40% is too low and therefor the seeded bug is too easy to find. This is supported by the fact that the violations already started to occur in the first few generations. This causes that a lot of encodings can find the bugs, meaning that a random initial population has a high probability of finding the seeded bug. Evolutionary algorithms are better against unguided approaches in finding difficult to detect bugs, as they are guided towards complex executions. This would mean that the evolutionary algorithms are currently not truly being evaluated as the bug is found before the evolutionary algorithm is really used. It could be useful to test our setups with a variety of seeded bugs with different difficulties of discovering them, for example with an agreement threshold of 60% instead of 40%. This way we could see if the complexity of the seeded bug is the reason for the similar performance of the experimental setups.

6.2.2 High mutation probability. Another possible reason could be that our mutation probability, which was 10% for every value within an encoding, was too high. Every encoding had a length of 294 so on average 29.4 values within an encoding would have been mutated. It is possible that these mutations caused the population to be very diverse for every generation. The evolutionary algorithms could possibly not have good exploitation using the fitness function because of this. Also the lack of elitism means that we could have lost good encodings when using crossover and mutation. This could be solved in future work by lowering the mutation probability or implementing elitism in the evolutionary algorithms.

7 Threats to Validity

In this section, we outline three main threats to the validity of our findings: nondeterminism in distributed systems, the re-presentability of our local testing environment, and the generalization of our results beyond the XRP Ledger Protocol (XRPL).

7.1 Nondeterminism

The nondeterministic nature of decentralized systems introduces inherent variability in the test results. Distributed systems are subject to concurrency bugs which can cause the same test case to give different results during multiple executions. We mitigated this partially by averaging the results of two iterations for each encoding, but this is insufficient to fully eliminate the effects of nondeterminism. Ideally, more repetitions would increase statistical confidence.

7.2 Re-presentability of Local System

Although our experiments were conducted in a controlled environment, this local setup may not fully represent the dynamics of a real-world XRPL deployment. Factors such as hardware limitations, the XRPL nodes being configured as tiny and low transaction volumes reduce the comparability of our experiments with real world implementation. Because of this our results may not fully reflect how the protocol behaves in production environments.

7.3 Generalization

In this research we have only tested one consensus protocol, XRPL. This means that our results can not be directly generalized to different consensus protocols, as our results are possibly specific to the XRPL code base. However, Groot is not dependent on the framework that it is implemented with. This means that future research could use Groot on different frameworks in order to test different consensus algorithms.

8 Responsible Research

Our research is not influencing humans directly, but finding bugs in a system which processes a lot of transactions is of course an ethical implication. For instance bugs could provide security breaches or exploitation options for the one who found it. If we had found bugs in the current version of the XRPL protocol, we would have disclosed it responsibly with the help of our supervisors.

8.0.1 Global Influence. A different concern is the impact our tests have on the production version of XRP which is currently run globally by Ripple. To avoid such problems we have ran all our tests on a private XRPL network, meaning that we could not have influenced the global XRPL network in any way. This also makes sure that our tests are better reproducible as it is not affected by any real world events.

8.0.2 Reproducibility. The reproducibility and replicability of our tests have already been discussed in the Study Design section 4. You can find the used versions of the code base, docker images, logs and instructions in the reproduction package on Zenodo [22].

9 Conclusion and future work

During this research we tested the XRP Ledger protocol using three different setups. The three tested setups were a random baseline, Simulated Binary Crossover operator in combination with Gaussian mutation and the Laplace operator with MPT mutation. We tested these setups on a bug seeded version of the XRP LCP version 2.4.0 in which the agreement threshold was lowered from 80% to 40%. We evaluated these setups based on effectiveness, how many violations were found, and efficiency, how fast and how many violations were found in this generation.

9.0.1 Effectiveness. During these tests all three setups found agreement violations indicating that the seeded bug was found. Seeing that all three setups had around the same amount of violations, the evolutionary algorithms are not more effective than a random baseline at finding bugs in the XRP Ledger protocol. This was possibly caused by a noisy fitness function, which possibly did not work well with priority-based event representation. It could also have been caused by the choice of event representation, as in other research the priority-based event representation is also less effective than for instance delay-based.

9.0.2 Efficiency. In the efficiency metric, the evolutionary approaches were also similar with finding the first violations. We noticed that SBX-Gaussian was the worst, but only had a one generation difference with the baseline and Laplace-MPTM. This could have been caused by the seeded bug being too easy to find, therefor random initialization already had a high probability of finding the bug. The high mutation probability and lack of elitism could also have been the cause for these results.

9.0.3 Conclusion. From this we can not conclude that operators have an impact on the effectiveness or efficiency of evolutionary algorithm in finding bugs in the XRP Ledger protocol. However since this could have been influenced by our choice of parameters, configuration and setup. We can conclude that bugs are discoverable by evolutionary algorithms and that they do not perform worse than a random baseline.

9.0.4 Future Work. For future work multiple directions can be explored. There are possible solutions to discover if evolutionary operators do have an impact on the performance of evolutionary algorithms.

First of all, a larger variety of seeded bugs could be tested. If future research seeds more bugs with different levels of difficulty, than it could be discovered if the seeded bug that we used was indeed to easy to find. This could also be a way of testing the performance of operators on problem complexity.

Second of all, parameters such as population size, amount of generations and specific values of operators could be tuned to achieve better results. This would allow the same setup to be used and could for instance be done using machine learning, which could be an interesting direction.

As a last option, a different fitness function which is not time dependent could create better results. This could have a positive influence since priority-based event representation does not create large amounts of delays. Also complex executions can occur without large ledger validation times. This would better be measured by for instance proposal-fitness.

To expand the scope of Groot, some other directions could be explored.

Firstly, Groot could be used on different consensus algorithms to see if results on the XRP LCP can be generalized. This could provide a way for evolutionary algorithms to be tested on more variety of applications. This could also be used as a starting point to generalize other research on XRPL to different distributed systems.

Secondly, future work could look at using different event-representations. We only looked at priority based event representation, but more ways can be explored to manipulate the network to find bugs. One option would be to look at the content of the messages and use this to delay certain critical messages. Another option would be to instead of delaying messages, messages could also be duplicated and send at a later time.

References

[1] 2022. Assessment of risks to financial stability from crypto-assets. *Financial Stability Board*, (Feb. 16, 2022). Retrieved Apr. 27, 2025 from https://www.fsb.org/2022/02/assessment-of-risks-to-financial-stability-from-crypto-assets/.

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

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

[4] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. Cofi: consistency-guided fault injection for cloud systems. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 536–547. doi:10.1145/3324884.3416548.

[5] Kalyanmoy Deb and Ram Bhushan Agrawal. 1995. Simulated binary crossover for continuous search space. *Complex Syst.*, 9, 2. http://www.complex-systems.com/abstracts/v09%5C_i02%5C_a02.html.

[6] Kalyanmoy Deb and Debayan Deb. 2014. Analysing mutation schemes for real-parameter genetic algorithms. *Int. J. Artif. Intell. Soft Comput.*, 4, 1, 1–28. doi:10.1504/IJAISC.2014.059280.

[7] Kusum Deep and Manoj Thakur. 2007. A new crossover operator for real coded genetic algorithms. *Appl. Math. Comput.*, 188, 1, 895–911. doi:10.1016/J.AMC.2006.10.047.

[8] Alkis Gotovos, Maria Christakis, and Konstantinos Sagonas. 2011. Test-driven development of concurrent programs using concuerror. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*. Kenji Rikitake and Erik Stenman, (Eds.) ACM, 51–61. doi:10.1145/2034654.2034664.

[9] Ralf Huuck, Gerwin Klein, and Bastian Schlich, (Eds.) *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*, (2010). USENIX Association. https://www.usenix.org/conference/ssv10.

[10] Wishaal Kanhai, Ivar van Loon, Yuraj Mangalgi, Thijs Van der Valk, Lucas Witte, Annibale Panichella, Mitchell Olsthoorn, and Burcu Kulahcioglu Ozkan. 2025. Rocket: A system-level fuzz-testing framework for the XRPL consensus algorithm. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2025, Napoli, Italy, March 31 - April 4, 2025*. IEEE, 737–741. doi:10.1109/ICST62969.2025.10988979.

[11] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. Tom Conte and Yuanyuan Zhou, (Eds.) ACM, 517–530. doi:10.1145/2872362.2872374.

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

[13] R.A.E. Mäkinen, J. Periaux, and J. Toivanen. 1999. Multidisciplinary shape optimization in aerodynamics and electromagnetics using genetic algorithms. *International Journal for Numerical Methods in Fluids*, 30, 2, 149–159. Publisher: John Wiley & Sons Ltd. doi:10.1002/(SICI)1097-0363(19990530)30:2<149::AID-FLD829>3.0.CO;2-B.

[14] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-based controlled concurrency testing. *Proc. ACM Program. Lang.*, 4, OOPSLA, 230:1–230:31. doi:10.1145/3428298.

[15] Fakhra Batool Naqvi, Muhammad Yousaf Shad, and Saima Khan and. 2021. A new logistic distribution based crossover operator for real-coded genetic algorithm. *Journal of Statistical Computation and Simulation*, 91, 4, 817–835. eprint: https://doi.org/10.1080/00949655.2020.1832093. doi:10.1080/00949655.2020.1832093.

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

[17] G. Pavai and T. V. Geetha. 2019. New crossover operators using dominance and co-dominance principles for faster convergence of genetic algorithms. *Soft Comput.*, 23, 11, 3661–3686. doi:10.1007/S00500-018-3016-1.

[18] Jennifer Rexford and Emin Gün Sirer, (Eds.) *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, (2009). USENIX Association.

[19] David Schwartz, Noah Youngs, and A. Britto. 2014. The ripple protocol consensus algorithm. In Retrieved Apr. 8, 2025 from https://www.semanticscholar.org/paper/The-Ripple-Protocol-Consensus-Algorithm-Schwartz-Youngs/bff4ecdd2c40bb67abab8d49e99c81287a7b2810.

[20] Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. 2022. Discotest: evolutionary distributed concurrency testing of blockchain consensus algorithms. Retrieved June 22, 2025 from https://resolver.tudelft.nl/uuid:5ac105ac-f2d0-4891-8b20-f5caae141854.

[21] Martijn van Meerten, Burcu Kulahcioglu Ozkan, and Annibale Panichella. 2023. Evolutionary approach for concurrency testing of ripple blockchain consensus algorithm. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 36–47. doi:10.1109/ICSE-SEIP58684.2023.00009.

[22] [SW] Bryan Wassenaar, Reproduction Package Groot version 1.0.0, June 2025. doi:10.5281/zenodo.15665194, URL: https://doi.org/10.5281/zenodo.15665194.

[23] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. 2023. Randomized testing of byzantine fault tolerant algorithms. *Proc. ACM Program. Lang.*, 7, OOPSLA1, 757–788. doi:10.1145/3586053.