Survival of the Fittest: Evaluating Fitness Functions for Concurrency Testing on the XRPL Consensus Protocol

Atour Mousavi Gourabi[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: Atour Mousavi Gourabi
Final project course: CSE3000 Research Project
Thesis committee: Burcu Kulahcioglu Ozkan, Annibale Panichella, Mitchell Olsthoorn, Jérémie Decouchant

Survival of the Fittest: Evaluating Fitness Functions for Concurrency Testing on the XRPL Consensus Protocol

Atour Mousavi Gourabi

Delft University of Technology
Delft, The Netherlands

Abstract

Distributed systems, such as blockchains, can have bugs around edge-cases that are hard to detect or trigger. Previous publications have introduced guided-search testing approaches that are able to find edge cases more efficiently than through conducting a systematic and exhaustive search. In this paper, we compare the effectiveness of fitness functions in evolutionary testing frameworks. For this we evaluate time and proposal fitness. While evolutionary testing frameworks are not new to the domain of concurrency testing consensus algorithms, the impact of the fitness functions that underpin them remains poorly understood. We use the XRPL consensus algorithm as a case study to evaluate the fitness functions using the Rocket testing framework. For this, we make use of seeded versions of XRPL. All evaluated fitness functions have been able to detect the bugs we seeded in the source code of the XRPL consensus algorithm. We show the validity of various fitness functions in trying to find the bug and analyze effects in the interplay between the time and proposal fitness functions we examine.

1 Introduction

Since the introduction of Bitcoin in 2009, the importance of cryptocurrencies has grown significantly. This gave rise to many prominent cryptocurrency projects. One such example is Ripple, which relies on the XRP Ledger and the corresponding XRPL consensus protocol. This consensus protocol ensures all nodes in the network agree on a certain state for the ledger. For the proper functioning of this blockchain and more generally the entire financial ecosystem built on top of it, the correctness of this consensus protocol is of paramount importance. To illustrate this point, as of May 2025, Ripple is one of the largest cryptocurrencies around by market capitalization and boasts daily traded volumes in excess of US$2B. Any breaches to the integrity of the network, such as forks, where the network fractures into multiple conflicting, smaller sub-networks, would pose serious risks to the stability and integrity of these markets and the confidence investors and other market participants have in them.

As was visible during the fallout of the Celsius and FTX scandals, such *black swan* events have the potential to reverberate far beyond the affected project itself, into the wider cryptocurrency and financial markets. These concerns are shared by institutions, such as the Financial Stability Board, which claimed the possibility of exposure to system-wide risks through cryptocurrencies and implementation errors in blockchain systems. To avoid such problems from arising through mundane implementation errors in these systems and to ensure that the XRPL consensus algorithm which we use as a case study here delivers on the guarantees from the whitepaper by Schwartz et al. [27], it is imperative we have the methods and tooling to test the correctness of implementations of these systems.

The non-deterministic nature of distributed systems means that there are many valid orderings in which the messages can be sent or arrive [18]. When real-world delays are introduced, this number tends to grow quickly. When this happens, it rapidly becomes unmanageable for systematic and exhaustive testing approaches. The requirement of simulating the potentially large network also adds to the cost of testing such systems. To manage testing these systems, van Meerten et al. [34] introduces an evolutionary testing algorithm, which they show can effectively detect bugs in these systems, also using the XRPL consensus algorithm as a case study. Using this testing algorithm, they uncovered a previously unknown live bug in the implementation of the XRPL consensus algorithm. There are two main hyperparameters which are integral to evolutionary testing algorithms: firstly the manner through which the best cases are selected and secondly the strategy responsible for generating offspring. Though van Meerten et al. [34] did investigate the efficiency of two fitness functions as single-objective optimization functions, the precise impact that both of these parameters have on the efficacy of this evolutionary testing algorithm for testing these kinds of systems remains poorly understood. Especially multi-objective fitness functions remain underexplored.

To allow researchers to implement algorithms to test the XRPL consensus algorithm, Kanhai et al. [17] introduced the Rocket testing framework. On top of this framework, we introduce an implementation of the evolutionary testing algorithm that was devised by van Meerten et al. [34]. In doing so, we provide a number of different fitness functions for the guided evolutionary search that is central to the algorithm, which we then compare for their ability to find bugs as efficiently as possible and their ability to find as many bugs as possible. As mentioned, while it is known to impact the efficiency of the approach, exactly how the different fitness functions impact the effectiveness of this approach is not yet fully understood. This is the main contribution of our research. Fitness functions are tested and compared against one another using the same operators and systems under test. Their evaluation takes into account the efficiency with which they have been able to find bugs, as well as the number of bugs that the guided evolutionary search has been able to find using these fitness functions.

Our results indicate that it can be difficult for the evolutionary algorithm to operate on the surface it is optimizing over. This extends to all examined fitness functions and configurations. Multi-objective fitness configurations are greatly affected by this, making it hard to see any practical benefits of incorporating two fitness objectives into the search. Our results also show a great correlation between some of the fitness functions and the detection of

violations. This correlation is significant in all considered setups and shows that especially the time fitness we are optimizing for in some configurations poses a good objective for finding bugs. Our main contributions consist of the following:

(1) Replicate part of van Meerten et al. [34] and compare the time and proposal fitness functions' efficacy in evolutionary search.
(2) Examine the usage of multi-objective fitness functions to time and proposal fitness in evolutionary search.
(3) Compare the impact of time and proposal fitness on the detection of violations.
(4) Compare the relation of time and proposal fitness.

2 Background

2.1 XRP Ledger

The XRPL consensus algorithm [2, 4, 27] forms the backbone of the XRP Ledger. It ensures a common state of transactions between the nodes in the network. The algorithm is a Byzantine fault tolerant algorithm, which means that it can function even with malicious nodes in the network [19]. In doing so it provides four guarantees or consensus properties.

(1) **Agreement**, which states that all honest processes decide on the same state.
(2) **Integrity**, which states that all honest processes decide at most one time.
(3) **Termination**, which states that all honest processes will eventually decide and not loop infinitely.
(4) **Validity**, which states that honest processes can only decide on a state that was proposed by an honest process.

The routine of messages that are sent between the nodes in the network is a cornerstone of the consensus algorithm. These messages are sent in three stages. To agree on the new set of transactions that are going to be added to the ledger, first a set of transactions is proposed. After initial agreement on a set of new transactions is reached in order to discard poorly supported transactions, the nodes will attempt to validate the agreement on the transactions to ensure they are the same. After the set of new transactions is validated, they are effectuated on the ledger and the new ledger state with the new transactions embedded is validated once more. After this, the cycle continues. The exact message types related to reaching consensus are presented in Table 1.

In the way the XRPL network sends and receives messages lies an inherent non-determinism. As the XRPL network is distributed, there will often be delays in when messages are sent or received. This can be due to any number of both internal and external processes interfering. Think of delays in the internet connection, partial outages, nodes dealing with performance problems, and so on.

2.2 Evolutionary Testing

Authors such as Jones et al. [16], Tracey et al. [32] introduced evolutionary algorithms to software testing in the 1990s. This quickly spawned efforts creating tools such as EvoSuite [12]. These algorithms run cycles which create and evaluate test cases. They do this in roughly four stages, which repeat as in Figure 1.

Table 1: XRPL message types related to reaching consensus.

Message Type	Description
`GetLedger`	Fetches missing ledger data from peer nodes.
`HaveTransactionSet`	Sent to signal that a transaction set has been obtained.
`LedgerData`	Fulfills a request for ledger data.
`ProposeSet`	Proposal of a set of transactions from a node for the next ledger.
`StatusChange`	Sent to confirm that a node closes its current ledger or accepts a new one.
`Transaction`	Sent to submit a transaction to be included.
`Validation`	Sent to confirm a node agrees on the validity of the ledger.

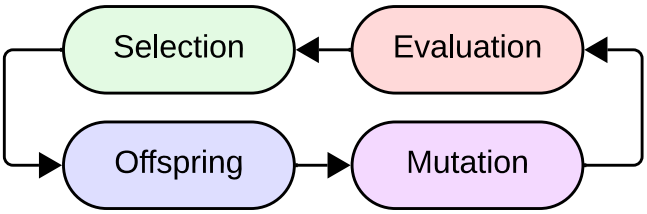

Figure 1: Evolutionary testing cycle

(1) In the **evaluation** stage the test case is evaluated. It is assigned a score by a fitness function. This score will then be used as the heuristic to guide the search for the best test cases. One such fitness function is time fitness, where the test cases are selected on the basis of how short or long it took for them to execute.
(2) Then, during **selection** the results from the evaluation are used to select the best test cases. Roughly speaking there are two main ways to do this, deterministically and probabilistically. Deterministic methods always select the test cases which they consider most optimal, a basic example of this is elitism, which always selects the test case with the highest fitness value. Alternatively, probabilistic selection procedures exist. These pick the test cases with some degree of randomness. An example of this is the roulette selector, which proportionally selects between the test cases based on their fitness value. A test case where $f = 5$ will be 5× more likely to be selected than a test case with $f = 1$.
(3) After selecting the best test cases, we must generate new test cases. This is done by combining them to create a new generation, the **offspring**.
(4) After creating the next generation, we must ensure that we are adequately equipped to explore the entire search space. To do this a random effect, the **mutation**, is used to distort the test case.

The application to the testing of blockchains by van Meerten [33] showed its potential in this domain.

2.3 Alternative Testing Approaches

Next to the guided evolutionary search for test cases, other approaches exist. Some alternative tooling, such as dBug [28], MoDist [36], Concuerror [14], DeMeter [15], SAMC [20], and FlyMC [23], makes use of systematic, exhaustive search. While they make use of optimizations to reduce their search space, it still grows at an incredible pace [11]. Because of this, they are impractical for large systems such as blockchains.

A more practical alternative for large systems is random testing. Tools such as CoFi [5] implement this method. Thomson et al. [29] showed that this performs better than exhaustive approaches for testing real-world applications. These experimental results have been backed up by theoretical explanations since then [24]. Later, van Meerten et al. [34] showed that guided evolutionary search outperforms random testing approaches.

A more recent alternative is ByzzFuzz [35], which makes use of a malicious node in the test network. By showing that it was able to uncover critical bugs in the XRPL v1.7.2 source code,[1] ByzzFuzz became an example that randomized testing is able to trigger edge cases. However, all in all it seems that system-level testing on blockchain systems remains underexplored [30, 31].

3 Methodology

Like van Meerten et al. [34], we make use of a delay based representation of events. This means that our test cases for the network consist of a collection of delays for the messages that are sent between the nodes. Adding these delays can cause the ordering between messages to be drastically altered in a distributed system such as the XRPL consensus algorithm, where only a partial ordering can be ensured [18]. Our testing approach seeks to exploit this property, by injecting these delays to trigger buggy, edge case behavior by the system. By using the evolutionary search, we aim to avoid having to perform an exhaustive search for bugs, which would be impractical for a system like this.

It is known that for certain optimization problems, smaller population sizes can be preferred [6, 7]. Our problem is a great candidate for this, as the evaluation of our test cases is fairly expensive. Due to this, van Meerten et al. [34] used a population size of 8. In our experiments we are using a population size of 10, this number is somewhat arbitrary, but low enough given the cost of running our evaluation. This means that compared to van Meerten et al. [34] we allow for slightly more exploration and diversity to be maintained during the run. During the experiment, we evaluated the evolutionary algorithm for 50 generations per run. This is to essentially guarantee convergence on a detection of the violation if it exists.

We used the structure outlined in Figure 2, using the Rocket testing framework [17] and Docker [25] to run the experiments. Every iteration was run using a separate Rocket instance, which managed spawning the test network. Our evolutionary test manager performed the test case selection and mutations. The Docker daemon was shared between all Rocket instances in the experiment through a mounted socket.

[1] https://xrpl.org/blog/2021/rippled-1.7.2.

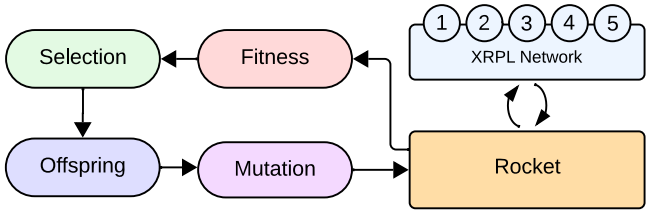

Figure 2: Experimental setup for evolutionary testing strategies using Rocket

During the experiments, we run the algorithm over a bug seeded version[2] of the latest version of the XRPL consensus algorithm, which is v2.4.0 at the time of writing.[3] We do this to ensure that there are valid violations to analyze and such that we know what to look for during our analysis of the runs. The bug that we seeded relates to the agreement process. Instead of the default threshold of 80% on agreement, we lowered this to 40% in our version of the algorithm. This seeks to trigger violations of the agreement property, which we can easily detect through the logging functionality in Rocket. In part, we are also almost reproducing part of the work done by van Meerten et al. [34], who analyzed a similar seeded bug.

Prior research has made use of transactions that are structured from three accounts, where account one overspends by concurrently sending 2×2 identical transactions to two different accounts. 80 000 is sent from account one to account 2 twice at two different validator nodes and 80 000 is sent from account one to account three twice as well at two more validator nodes. This setup is heavily inspired by van Meerten et al. [34].

To run our experiments, we are using a $(\mu + \lambda)$ evolutionary algorithm, much like van Meerten et al. [34]. The outline of this evolutionary algorithm is provided as pseudocode in Algorithm 1.

Algorithm 1 Pseudocode for a $(\mu + \lambda)$ evolutionary algorithm, adapted from van Meerten et al. [34]

```
1:  procedure EvolutionaryAlgorithm(μ, λ)
2:      parents, offspring ← Initialize(μ, λ)
3:      while t < 100 do
4:          Evaluate(offspring)
5:          parents ← Select(parents + offspring)
6:          offspring ← Recombine(parents)
7:      end while
8:  end procedure
9:
10: function Recombine(parents, λ)
11:     offspring ← Crossover(parents, λ)
12:     for individual ∈ offspring do
13:         MutateGaussian(individual)
14:     end for
15:     return offspring
16: end function
```

[2] https://github.com/amousavigourabi/bug-seeded-rippled/tree/seeded-2.4.0-fully-lowered-threshold.
[3] https://xrpl.org/blog/2025/rippled-2.4.0.

We are examining the impact of the fitness and selection routines during a run of the evolutionary algorithm. For this, we use time fitness and proposal fitness using elitist, roulette, and tournament-based selection methods in specific configurations which are outlined in section 4.

(1) **Time fitness** will guide the search towards test cases that take the longest to run. The idea behind this is that if we keep selecting for test cases that take longer and longer to execute, we might at some point reach an edge case that will exhibit non-compliant behavior on properties, such as termination.
(2) **Proposal fitness** will guide the search towards test cases with higher proposal sequences. The idea behind this is that if we keep selecting for test cases with higher and higher proposal sequences, we might reach edge cases that could break the consensus properties.

The selection methods we examine are more diverse. Like van Meerten et al. [34], we examine elitist selection. In addition to this, we expand our search by looking into the roulette [22] and 4-way tournament [13] selection procedures for single-objective optimizations over both proposal and time fitness. We also add two multi-objective selection procedures, namely the dominance and crowding distance-based tournament [8] and NSGA-II [9].

(1) **Elitist** selection always ensures to pick only the 'fittest' individuals. It never selects individuals which score worse on the fitness function. This is constrained to single-objective optimization, meaning that we use two configurations, one to test it with time fitness, and one to test it with proposal fitness.
(2) **Roulette** selection picks the cases based on their fitness functions in a proportional, probabilistic way. If a test case is five times as good as another, the odds that it is selected will be five times greater. This is constrained to single-objective optimization, meaning that we use two configurations, one to test it with time fitness, and one to test it with proposal fitness.
(3) **4-way tournament** selection is based on a series of fully random head-to-head comparisons. These comparisons are performed in a tournament of size four. In these tournaments, the test cases are compared on their fitness, where the best prevails. Unlike its dominance-based counterpart, the dominance and crowding distance of a test case have no direct influence on the outcome of this selection procedure. For this selection procedure, we use two configurations, one to test it with time fitness, and one to test it with proposal fitness.
(4) **Dominance-based tournament** selection is based on a series of fully random head-to-head comparisons. Here, a fully randomized set of test cases are compared on their fitness, where the best prevails. This comparison is based on the dominance of the test case and if there is a tie there, the crowding distance. This is done to ensure diversity when neither dominates the other.
(5) **NSGA-II** selection is based on non-dominated sorting and a crowding distance metric. Test cases are ranked by their Pareto dominance. Within each level, the metric for crowding distance is used to ensure diversity. Selection favors individuals from better fronts and less crowded regions.

For the mutation and recombination sections of the evolutionary algorithm, we make use of the simulated binary crossover and a Gaussian mutation. We set the simulated binary crossover with parameter $\eta = 3.0$, just like van Meerten et al. [34]. The Gaussian mutation [10] is set with parameters $\mu = 0$ and $\sigma = 40$, and a mutation probability of 0.1 per item.

To ensure reproducibility, our results and experimental setup are provided in the reproduction package.[4] This will allow others to rerun our experiments. Reproduction is impacted by nondeterminism in our experiment, as we execute only one run of the genetic algorithm and it is as such not fully robust to the nondeterministic effects that are inherent to the method.

4 Study Design

The purpose of this study is chiefly experimental reproduction, with additional benchmarking of unexplored strategies. The outcomes of the experiment are evaluated quantitatively. To evaluate the use of fitness functions and selection procedures in evolutionary testing on the XRPL consensus algorithm we answer the following research questions.

(1) *How effectively does our testing approach discover bugs in XRPL consensus algorithm implementations?*
(2) *How does the bug detection performance of the algorithm using delay based representations compare to a random baseline algorithm?*
(3) *How does the selection procedure for test cases with delay based representations affect the performance of the evolutionary algorithm?*
(4) *How good of an objective are the time and proposal fitness functions for finding bugs?*

The evaluation of these four research questions will allow us to answer our main thesis, which states: *"How efficient are different fitness and selection procedures during the evolutionary algorithm in testing the XRPL consensus algorithm using delay based representations?"* This will create a holistic evaluation of the efficiency of the different approaches to fitness and selection using evolutionary testing that were investigated.

As described in Table 2, we evaluated nine approaches within the evolutionary testing framework from van Meerten [33] and an independent random baseline.

To answer research question one, we run all the specified configurations on the bug seeded version of XRPL. These runs are analyzed and checked for any runs where one of the consensus properties fails, which potentially signals bugs. If these cannot be found we stop here, if these can be found we try to uncover what causes the error and report this using responsible disclosure.

To answer research question two, the configurations are then compared with the baseline on the basis of whether they could find the bug and how fast they managed to find it. If the configuration found the bug faster than the baseline did and a contingency

[4] https://github.com/amousavigourabi/rocket/tree/reproduction-package-atour.

Table 2: Fitness function and selector configurations during the experiments

Case	Selection	Fitness
Baseline	Random	None
Elitist time	Best	Time fitness
Elitist proposal	Best	Proposal fitness
Roulette time	Roulette	Time fitness
Roulette proposal	Roulette	Proposal fitness
Tournament time	4-way tournament	Time fitness
Tournament proposal	4-way tournament	Proposal fitness
DCD Tournament	Dominance-based tournament	Time and proposal fitness
NSGA-II	NSGA-II	Time and proposal fitness

test signals a statistically significant p-value on the amount of violations that could be caught, we must conclude the bug detection performance of the evolutionary algorithm using this configuration is superior to the random baseline.

To answer research question three, we can build upon the experiment for research question two. Here, we will compare the configurations against one another instead of the baseline. A configuration will be superior when it finds a bug that the other configuration fails to find, or when it is able to find it faster than the other configuration.

To answer research question four, we build upon the results from research questions one to three. Here, we analyze the efficacy of the evolutionary approach in finding bugs. The overall results of how often the bugs can be found is analyzed. Subsequently, this analysis is supplemented by existing benchmarks, with which we will compare the performance of our configurations that are under test. This will allow us to assess whether our configurations pose any practical benefits compared to other testing approaches.

The experiments are run on a server node from the Software Engineering Research Group at the Delft University of Technology. The node was shared with four other research efforts during the experiments and boasts 2× AMD EPYC 7H12 64-core (128-thread) processors at 1.6–2.6Ghz, with a total of 256GB of RAM. To avoid interference on our experiments by the other research efforts using the same computational resources, we deliberated and divided the resources on the node beforehand. This guaranteed the node would not suffer from out of memory errors during the experiments and that the experiments would not be harmed or compromised in other ways, such as through the test networks not booting properly due to a lack of resources.

During our experiments we have observed test case runtimes of around 290 seconds. In our 9 configurations of 50 generations with 1 iteration per test case and a population size of 10, this has resulted in a sequential runtime of 9×290×10×1×50 = 1 305 000 seconds = 362 hours and 30 minutes.

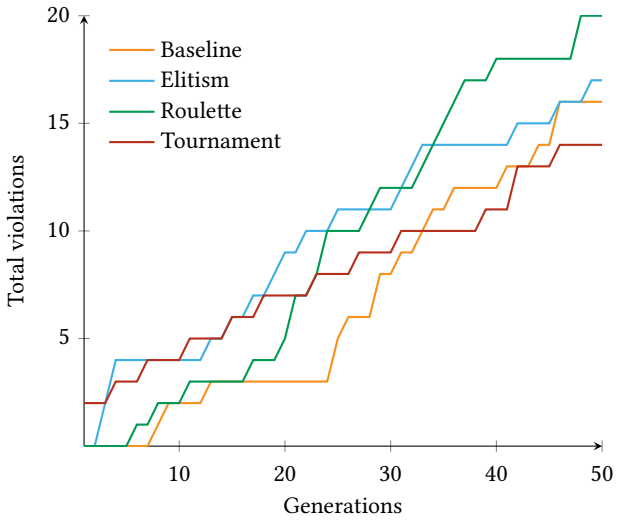

Figure 3: The cumulative amount of detected violations at each generation by each configuration using time fitness on the bug seeded version of XRPL v2.4.0

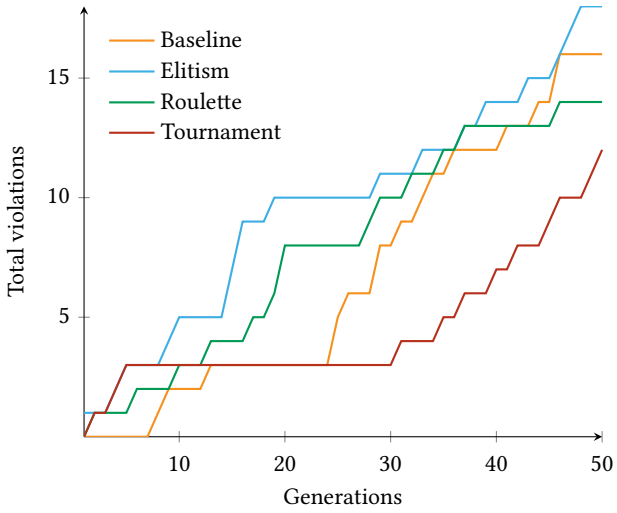

Figure 4: The cumulative amount of detected violations at each generation by each configuration using proposal fitness on the bug seeded version of XRPL v2.4.0

5 Results

We present our results by showing the graphs of the cumulative detected violations by configuration, looking at the convergence of the fitness values in the runs, and examining the utility of the fitness functions for finding test cases that show violations.

As shown by Figure 3–5, all test cases are able to find violations from the seeded bug. It can also immediately be observed that they do not perform much better than the random baseline. In Figure 6 we can see that the evolutionary algorithm has an incredibly hard time converging to its objectives under all tested configurations.

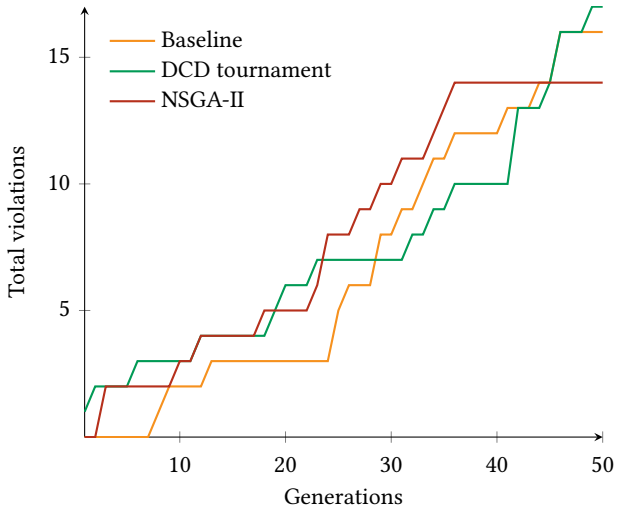

Figure 5: The cumulative amount of detected violations at each generation by each multi-objective fitness configuration on the bug seeded version of XRPL v2.4.0

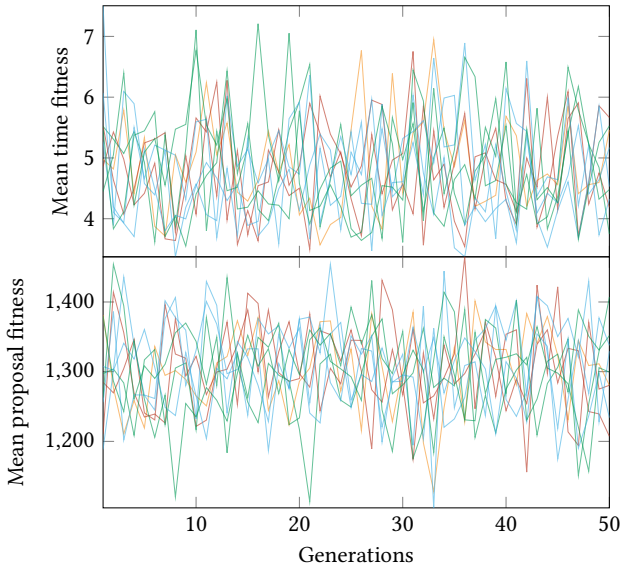

Figure 6: The mean time and proposal fitness of each generation by configuration on the bug seeded version of XRPL v2.4.0

Over the 50 generations that we ran the algorithm for, it has mainly been white noise. As can be seen in Table 3, all but one of the time series does not make the $p < 0.05$ threshold and has statistical support against stationarity. Even the only configuration which does not obviously seem stationary only has a resulting p-value of 0.08, given the sample size of 2×9 configurations and fitness functions, this signal seems spurious. We can infer that the algorithm does not manage to converge for any of our configurations.

Table 3: Augmented Dickey–Fuller test results for stationarity per fitness and configuration, p-values are provided up to two decimals

Case	Time	Proposal
Baseline	8.73e-08	2.87e-08
Elitist proposal	1.86e-10	0.02
Elitist time	8.49e-10	0.01
Roulette proposal	7.21e-07	6.38e-06
Roulette time	5.73e-10	0.08
Tournament proposal	1.81e-08	5.8e-08
Tournament time	8.7e-16	2.92e-09
DCD tournament	0.034	1.29e-10
NSGA-II	1.8e-4	2.79e-09

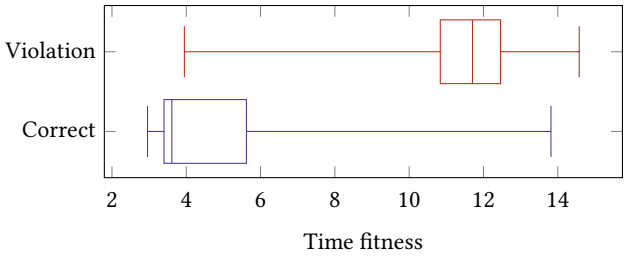

Figure 7: The time fitness of all runs summarized per test case outcome on the bug seeded version of XRPL v2.4.0

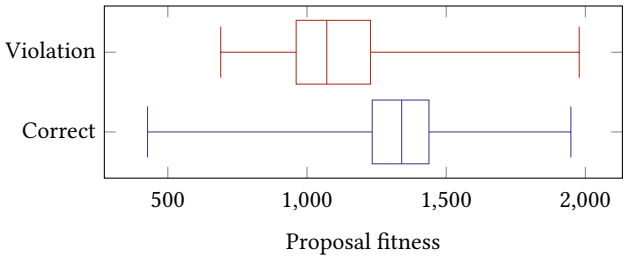

Figure 8: The proposal fitness of all runs summarized per test case outcome on the bug seeded version of XRPL v2.4.0

In Figure 7 and Figure 8, we present the time and proposal fitness values for all runs, grouped by whether they contained any detected violations. The box plots show clearly a great difference between the time fitness values between the correct runs and the edge case runs where we manage to detect the bug. For the proposal fitness a similar, but reversed effect also seems to exist, be it less pronounced. To test the relationship between the time and proposal fitness and the ability of the algorithm to detect bugs, we used the point biserial correlation test [21]. Formally in our test, the trait is the detection of a violation and our statistic is the time fitness or proposal fitness. As can be seen in Table 4, the p-values indicating a relationship are incredibly well supported for time fitness and also receive support for proposal fitness on most configurations. The overall p-values are well within the threshold for statistical

Table 4: Correlation coefficients and p-values between time and proposal fitness and the detected violations in each configuration, rounded to two digits

	Time		Proposal	
Case	**p-value**	r	**p-value**	r
Baseline	7.19e-25	0.44	5.46e-5	-0.18
Elitist proposal	5.77e-37	0.53	8.33e-8	-0.24
Elitist time	9.37e-34	0.51	4.4e-4	-0.16
Roulette proposal	2.09e-24	0.43	0.03	-0.1
Roulette time	2.98e-34	0.51	2.77e-3	-0.13
Tournament proposal	3.47e-16	0.35	0.08	-0.08
Tournament time	6.52e-33	0.50	0.05	-0.09
DCD tournament	5.22e-23	0.42	0.02	-0.11
NSGA-II	3.41e-29	0.47	1.71e-9	-0.27
Overall	3.14e-238	0.46	3e-24	-0.15

Table 5: Significance values of outperforming the random baseline.

Case	**p-value**
Elitist time	0.43
Elitist proposal	0.37
Roulette time	0.25
Roulette proposal	0.65
Tournament time	0.65
Tournament proposal	0.78
DCD tournament	0.43
NSGA-II	0.65

significance. These results suggest that especially time fitness is a good heuristic for finding agreement violations.

6 Discussion

To answer our first research question: *"How effectively does our testing approach discover bugs in XRPL consensus algorithm implementations?"* We now know that the best of our configurations, roulette time, can find 20 bug-related violations over 50 generations with the seeded algorithm implementation. This means that we can find the seeded bug and that we can find the seeded bug relatively fast.

Then, in order to answer our second research question: *"How does the bug detection performance of the algorithm using delay based representations compare to a random baseline algorithm?"* We must perform single-tailed hypothesis testing. When considering the resulting observations within a model as independent Bernoulli trials, we can calculate p-values for the hypothesis that one strategy beats the baseline. This is done through the hypothesis that $p_{\text{strategy}} > p_{\text{baseline}}$, where p is the probability for the underlying Bernoulli trial. When this is tested for using Boschloo's test [3], we get the results in Table 5. Our Bernoulli trials are likely to be autocorrelated through the usage of the evolutionary algorithm, which can lead to overly optimistic p-values [1]. We can thus potentially see the results of these tests as some sort of lower bound.

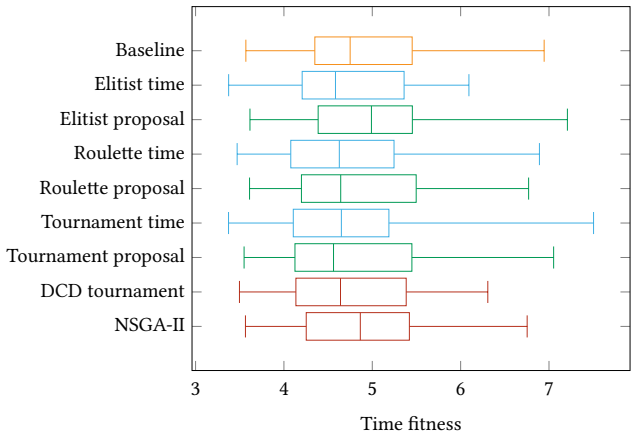

Figure 9: The time fitness summarized per configuration on the bug seeded version of XRPL v2.4.0

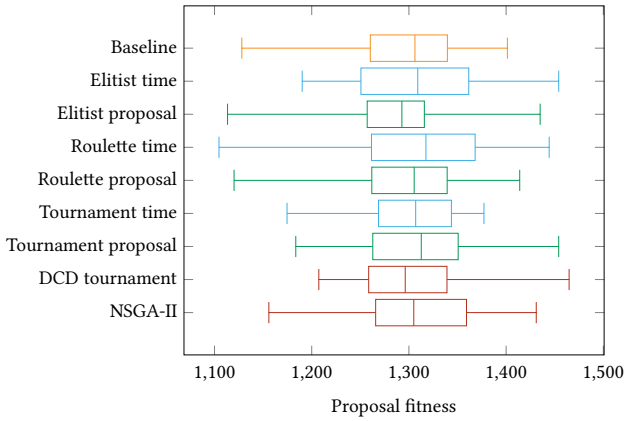

Figure 10: The proposal fitness summarized per configuration on the bug seeded version of XRPL v2.4.0

The tests clearly show that the p-values are never significant, making it unlikely that our setup with the specific mutation procedure in section 3 outperforms the random baseline.

Given the prior results that we have from van Meerten et al. [34], who did show a significant result on this front, this is interesting. It is more than possible that this is due to our experimental setup. Unlike van Meerten et al. [34], we only made use of one run of 50 generations. On the other hand, van Meerten et al. [34] used 30 different runs. Next to that, van Meerten et al. [34] also uses different settings for the mutation operator. Our results can thus not be mapped onto one another one-by-one. Our findings suggest that the surface over which the evolutionary algorithm is trying to optimize could be different from what our selected mutation operator can handle properly. It might be either too explorative or too exploitative in this regard. Another possibility is that our use of only 1 iteration during testing makes our observed time or proposal fitness too unreliable to be a good estimator that we can optimize over. This would explain the stationarity on the fitness functions we showed in Figure 6.

To answer our third research question: *"How does the selection procedure for test cases with delay based representations affect the performance of the evolutionary algorithm?"* We can for now, given the uniform stationarity in our results as in Figure 6, say that this cannot be answered yet. We already know the runs in the experiment are stationary with regards to the fitness functions and in Figure 9 and Figure 10 we see that the ranges of values per configuration are also roughly consistent.

For the multi-objective functions, we can do some meagre analysis on the basis of the data presented in Figure 8 and Figure 7. Next to these relationships between the time and proposal fitness on the one hand and the detection of a violation on the other, there is namely also a relationship between the time and proposal fitness values in our sample. Their Pearson correlation [26] reveals a p-value of under 1e-16 and a strong correlation coefficient of ≈ -0.63. What this means for our multi-objective fitness functions is that they are essentially trying to optimize two inversely correlated metrics in the same direction. While optimizers such as NSGA-II are designed for use with potentially conflicting objectives, this might hamper their usability.

For our fourth research question: *"How good of an objective are the time and proposal fitness functions for finding bugs?"* we can see that especially both fitness functions can be a good objective for the type of seeded bug that we examined. As we can see in Table 4, there is a very strong relationship between the time fitness of the test cases in the analyzed runs and whether we detect a violation. A larger time fitness indicates a higher chance of a violation. Surprisingly, proposal fitness has an inverse relationship. Though the relationship is less pronounced, we can safely say it is statistically significant.

We can first try to explain the performance of proposal fitness as a predictor through the observed inverse correlation with time fitness. However, when controlling for the correlation between the time and proposal fitness, the residual proposal fitness remains a statistically significant factor under the point biserial correlation test, with a p-value of 2.91e-32 and a coefficient of ≈ 0.17. This means that the explanation of the impact of proposal fitness on the detection of the bug is not solely through its observed relation to time fitness.

7 Threats to Validity

We used the tiny configuration for XRPL nodes. These configurations are not used in production, this means that our test network is not an exact replicated version of the network as it is in production. While these tiny nodes do behave like production nodes, they are resource bound and keep track of less data. This choice was made due to resource constraints, it will be functionally impossible for the study to be reproduced if nodes are run with production settings. We do not expect this decision to hurt our results, but we cannot guarantee this to hold.

Only one iteration is used during the evaluation of our test cases. Due to the non-deterministic nature of the system under test, this means that our metrics for evaluating the test cases, used for the fitness calculation and selection procedure, are not and simply cannot be robust. This is well below the amount required for good confidence that the results are able to capture edge behavior if it exists. As this study focuses on the impact of the fitness function and selection procedures, which are impacted significantly by this, it is an important limitation to the results.

As we use the XRPL consensus algorithm as a case study, there is a risk that the results of our experiment do not generalize to other similar blockchains. There are no theoretical hindrances to generalizing the core results to other Byzantine fault tolerant consensus algorithms. Further research will be necessary to confirm this.

The experiments we have run measure the impact of the fitness function and selection procedure for one seeded bug. It is trivial that the optimization surface for other, potentially live, bugs looks different than our seeded one. This means that it is possible for this surface to look drastically different to the one analyzed here. It is likely that our analysis of properties such as the smoothness of the surface do not generalize to all other bugs. This will impact the validity of our results in a more generic setting, where certain selection procedures might be a better fit for the concrete circumstances of the bug that is found. We encourage more research into this to evaluate the generalizability of our results and the extent of the potential heterogeneity that might be observed in these surfaces.

The non-determinism involved in the execution of our experiments also poses a threat. It is possible that our executions may have been outliers. In order to mitigate this, further research may choose to use methods where many more runs are compared to one another. One such method is used by van Meerten et al. [34], where many runs over a few configurations are compared on the binary measure whether or not they manage to find a specific bug. This is a more robust measure, which we encourage further research to examine.

8 Conclusion and Future Work

We found that the random baseline does not yield inferior performance to the configurations we examined for the evolutionary guided search using delay based representations. We showed that both time and proposal fitness are attractive optimization objectives. We explored some theoretical considerations around the usage of multi-objective optimization functions using these two objective functions.

Further research needs to be done in replicating our results in a more robust setting, with larger iteration sizes. The influence of the small iteration sizes could well have created a setting in which the evolutionary algorithm did not have access to reliable enough estimators for the fitness functions it is optimizing over. The influence of the fitness function in particular and the exploration of other fitness functions can be explored in more depth. An interesting question to ask could be whether our results for the efficacy of the fitness functions generalize beyond the agreement bug we examined, to become applicable in a broader context.

Through our exploration of the relationships between time and proposal fitness and their impact on finding the seeded bug, the results of van Meerten et al. [34] are provided with an explanatory basis for part of the effect of runs that are optimized over proposal fitness on finding the seeded agreement bug.

Responsible Research

During the experiment we made use of a test network that was managed by the Rocket testing framework, we did not involve the production network in our tests to ensure we would not harm stability. The goal of this experiment is to better understand the network and ensure it functions securely, we do not intend to exploit any bugs we find. All live bugs found will be submitted to the XRP Ledger development team using responsible disclosure.[5] Responsible disclosure is incredibly important in our case as it concerns a potentially live system that is being used as financial infrastructure. Any unpatched or unconfirmed public bug reports could have a grave impact on the ecosystem built on top of this technology. This would be able to cause irreparable damage.

To limit the ecological footprint of the experiment, we used tiny configurations of the XRPL nodes to construct the test networks. The experiments were performed using private compute infrastructure from the Software Engineering Research Group at Delft University of Technology. A reproducibility kit is provided to ensure others can reproduce and validate our results. No humans were harmed in conducting the experiments. Furthermore, we declare that we have no competing interests.

To allow for the reproduction of the experiments, a reproduction package is provided. As are the results and setup of the experiments. Some of the inherent non-determinism in executing test cases in the workings of the consensus protocol in the test network remain. These cannot really be accounted for as of yet next to large sample sizes and iterations, after which the law of large numbers will regulate the behavior of the collective. These kinds of experiments were outside of the scope of this research project.

References

[1] R. L. Anderson. 1954. The Problem of Autocorrelation in Regression Analysis. *J. Amer. Statist. Assoc.* 49, 265 (1954), 113–129.

[2] Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. 2015. Ripple: Overview and Outlook. In *Trust and Trustworthy Computing*, Mauro Conti, Matthias Schunter, and Ioannis Askoxylakis (Eds.). Springer, Cham, 163–180.

[3] R. D. Boschloo. 1970. Raised conditional level of significance for the 2×2-table when testing the equality of two probabilities. *Statistica Neerlandica* 24, 1 (1970), 1–9. doi:10.1111/j.1467-9574.1970.tb00104.x

[4] Brad Chase and Ethan MacBrough. 2018. Analysis of the XRP Ledger Consensus Protocol. arXiv:1802.07242

[5] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2021. CoFI: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '21)*. Association for Computing Machinery, New York, NY, USA, 536–547. doi:10.1145/3324884.3416548

[6] Tianshi Chen, Ke Tang, Guoliang Chen, and Xin Yao. 2012. A large population size can be unhelpful in evolutionary algorithms. *Theoretical Computer Science* 436 (2012), 54–70. doi:10.1016/j.tcs.2011.02.016

[7] Tinkle Chugh, Karthik Sindhya, Jussi Hakanen, and Kaisa Miettinen. 2019. A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms. *Soft Comput.* 23, 9 (2019), 3137–3166. doi:10.1007/s00500-017-2965-0

[8] Carlos A. Coello Coello and Efrén Mezura Montes. 2002. Constraint-handling in genetic algorithms through the use of dominance-based tournament selection. *Advanced Engineering Informatics* 16, 3 (2002), 193–203. doi:10.1016/S1474-0346(02)00011-3

[9] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2000. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In *Parallel Problem Solving from Nature PPSN VI*, Marc Schoenauer, Kalyanmoy Deb, Günther Rudolph, Xin Yao, Evelyne Lutton, Juan Julian Merelo, and Hans-Paul Schwefel (Eds.). Springer, Berlin, Heidelberg, 849–858.

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

[11] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.* 40, 1 (2005), 110–121. doi:10.1145/1047659.1040315

[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 416–419. doi:10.1145/2025113.2025179

[13] David E. Goldberg, Bradley Korb, and Kalyanmoy Deb. 1989. Messy Genetic Algorithms: Motivation, Analysis, and First Results. *Complex Syst.* 3, 5 (1989), 493–530.

[14] 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) *(Erlang '11)*. Association for Computing Machinery, New York, NY, USA, 51–61. doi:10.1145/2034654.2034664

[15] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 265–278. doi:10.1145/2043556.2043582

[16] Bryan F. Jones, Harmen-Hinrich Sthamer, and David E. Eyres. 1996. Automatic structural testing using genetic algorithms. *Software Engineering Journal* 11 (1996), 299–306. Issue 5. doi:10.1049/sej.1996.0040

[17] Wishaal Kanhai, Ivar van Loon, Yuraj Mangalgi, Thijs van der Valk, Lucas Witte, Annibale Panichella, Mitchell Olsthoorn, and Burcu Külahçıoğlu Özkan. 2025. Rocket: A System-Level Fuzz-Testing Framework for the XRPL Consensus Algorithm. In *18th IEEE International Conference on Software Testing, Verification and Validation (ICST) 2025* (Naples, Italy). IEEE Press, USA, 737–741.

[18] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565. doi:10.1145/359545.359563

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

[20] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI '14)*. USENIX Association, USA, 399–414.

[21] Joseph Lev. 1949. The Point Biserial Coefficient of Correlation. *The Annals of Mathematical Statistics* 20, 1 (1949), 125–126. doi:10.1214/aoms/1177730103

[22] Adam Lipowski and Dorota Lipowska. 2012. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications* 391, 6 (2012), 2193–2196. doi:10.1016/j.physa.2011.12.004

[23] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. doi:10.1145/3302424.3303986

[24] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.* 2, POPL, Article 46 (2017), 24 pages. doi:10.1145/3158134

[25] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J.* 2014, 239, Article 2 (2014), 1 pages.

[26] Karl Pearson. 1895. Note on Regression and Inheritance in the Case of Two Parents. *Proceedings of the Royal Society of London* 58 (1895), 240–242.

[27] David Schwartz, Noah Youngs, and Arthur Britto. 2014. *The Ripple Protocol Consensus Algorithm.* Technical Report. Ripple Labs Inc.

[28] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification* (Vancouver, BC, Canada) *(SSV '10)*. USENIX Association, USA, 3.

[29] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2016. Concurrency Testing Using Controlled Schedulers: An Empirical Study. *ACM Trans. Parallel Comput.* 2, 4, Article 23 (2016), 37 pages. doi:10.1145/2858651

[30] Marios Touloupou, Klitos Christodoulou, Antonis Inglezakis, Elias Iosif, and Marinos Themistocleous. 2021. Towards a Framework for Understanding the Performance of Blockchains. In *2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)* (Paris, France). IEEE Press, USA, 47–48. doi:10.1109/BRAINS52497.2021.9569810

[31] Marios Touloupou, Klitos Christodoulou, Antonios Inglezakis, Elias Iosif, and Marinos Themistocleous. 2022. Benchmarking Blockchains: The case of XRP Ledger and Beyond. In *55th Hawaii International Conference on System Sciences,*

[5] https://xrpl.org/blog/2025/rippled-2.4.0#bug-bounties-and-responsible-disclosures.

HICSS 2022 (Maui, Hawaii). ScholarSpace, USA, 1–8.

[32] N. Tracey, J. Clark, K. Mander, and J. McDermid. 1998. An automated framework for structural test-data generation. In *Proceedings 13th IEEE International Conference on Automated Software Engineering* (Honolulu, Hawaii) *(ASE '98)*. IEEE Press, USA, 285–288. doi:10.1109/ASE.1998.732680

[33] Martijn van Meerten. 2022. *DiscoTest: Evolutionary Distributed Concurrency Testing of Blockchain Consensus Algorithms*. Master's thesis. Delft University of Technology. https://resolver.tudelft.nl/uuid:5ac105ac-f2d0-4891-8b20-f5caae141854

[34] Martijn van Meerten, Burcu Külahçıoğlu Özkan, and Annibale Panichella. 2023. Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '23)*. IEEE Press, Melbourne, Australia, 36–47. doi:10.1109/ICSE-SEIP58684.2023.00009

[35] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Külahçıoğlu Özkan. 2023. Randomized Testing of Byzantine Fault Tolerant Algorithms. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 101 (2023), 32 pages. doi:10.1145/3586053

[36] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA) *(NSDI '09)*. USENIX Association, USA, 213–228.

Received 2 June 2025; revised 22 June 2025; accepted 27 June 2025