2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 979-8-3503-0037-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-SEIP58684.2023.00009

# Evolutionary Approach for Concurrency Testing of Ripple Blockchain Consensus Algorithm

Martijn van Meerten
*Department of Software Technology*
*Delft University of Technology*
Delft, the Netherlands
m.c.vanmeerten@student.tudelft.nl

Burcu Kulahcioglu Ozkan
*Department of Software Technology*
*Delft University of Technology*
Delft, the Netherlands
b.ozkan@tudelft.nl

Annibale Panichella
*Department of Software Technology*
*Delft University of Technology*
Delft, the Netherlands
a.panichella@tudelft.nl

***Abstract*—Blockchain systems are prone to concurrency bugs due to the nondeterminism in the delivery order of messages between the distributed nodes. These bugs are hard to detect since they can only be triggered by a specific order or timing of concurrent events in the execution. Systematic concurrency testing techniques, which explore all possible delivery orderings of messages to uncover concurrency bugs, are not scalable to large distributed systems such as blockchains. Random concurrency testing methods search for bugs in a randomly generated set of executions and offer a practical testing method.**

**In this paper, we investigate the effectiveness of random concurrency testing on blockchain systems using a case study on the XRP Ledger of the Ripple blockchain, which maintains one of the most popular cryptocurrencies in the market today. We test the Ripple consensus algorithm of the XRP Ledger by exploring different delivery orderings of consensus protocol messages. Moreover, we design an evolutionary algorithm to guide the random test case generation toward certain system behaviors to discover concurrency bugs more efficiently. Our case study shows that random concurrency testing is effective at detecting concurrency bugs in blockchains, and the evolutionary approach for test generation improves test efficiency. Our experiments could successfully detect the bugs we seeded in the Ripple source code. Moreover, we discovered a previously unknown concurrency bug in the production implementation of Ripple.**

***Index Terms*—Ripple, blockchains, distributed systems, consensus, concurrency, evolutionary algorithms, software testing**

## I. Introduction

After the introduction of Bitcoin in 2008, cryptocurrencies have been becoming mainstream. Financial Stability Board states that fast-evolving crypto-assets markets could reach a point where they represent a threat to global financial stability due to their scale, structural vulnerabilities, and increasing interconnectedness with the traditional financial system [1]. Increasing dependence on crypto-assets' financial stability emphasizes the need for reliable and robust technologies to manage them.

Blockchain is the driving technology behind Bitcoin and most other cryptocurrencies that followed. It allows for decentralization as it does not require a central trusted authority to execute and monitor transactions. A blockchain is a form of a distributed ledger where details on transactions and accounts are stored in blocks that are chained mathematically. A blockchain system consists of a network of nodes replicating a service and issuing client transactions on the blockchain ledger. The nodes participating in the network must agree on the transactions contained in a block and the total order of blocks in the blockchain.

The core of a blockchain system is the consensus mechanism that provides distributed agreement among the nodes in the blockchain network. The nodes use a consensus algorithm (a.k.a. consensus protocol) to validate transactions, group them into blocks, and chain them into a totally ordered set of blocks. The consensus protocol specifies rules for exchanging and processing messages between the nodes to achieve agreement on the state of the blockchain. The correctness of the consensus mechanism is crucial for the blockchain nodes to commit and append the same blocks of client transactions in the blockchain. Incorrect design or implementation of consensus mechanisms can cause forks in the network, leading to serious reliability and security problems. For example, consensus violations can manifest as an inability to issue any transactions or issuing conflicting transactions, resulting in security problems such as double-spending. Finding bugs before the deployment of systems is critical, especially for blockchain applications with immutable blocks of issued transactions.

As with other distributed systems, blockchains are prone to concurrency bugs due to the nondeterminism in the delivery order and timing of the messages in an execution. In executing a distributed system, the participating nodes execute at arbitrary speeds and only synchronize and communicate through exchanging asynchronous messages. The concurrency nondeterminism gives rise to many possible delivery orderings of the messages. Depending on the processing order of the protocol messages, the nodes can result in different possible internal states. Unexpected orderings of message delivery can bring the system into unexpected state changes, which might result in concurrency bugs [2], [3].

This paper describes a case study for concurrency testing of Ripple, the large-scale enterprise blockchain application for global payments. At the time of this submission, Ripple's global payments network includes over 300 leading partners in the finance sector in 40+ countries, six continents and reached over 1.29 million transactions per day. This makes Ripple a unique case study for assessing the effectiveness and scalability of concurrency testing approaches.

2832-7659/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-SEIP58684.2023.00009

We test the Ripple XRP Distributed Ledger to search for concurrency bugs in the Ripple Consensus Algorithm (RCA), which guarantees agreement on the next ledger to append to the Ripple blockchain. Although RCA runs in a distributed setting with concurrency nondeterminism, its design and implementation heavily rely on a common notion of time and synchronous execution of the nodes in the Ripple network. Our case study explores whether RCA continues to satisfy agreement in subtle concurrency scenarios with different delivery orderings and timings of the messages.

We use two approaches for the concurrency exploration of RCA. First, motivated by the empirical evidence for the success of random concurrency testing [4], we investigate the effectiveness of two random concurrency testing methods to search for concurrency bugs in blockchain systems: Delay-based random testing ($RandomDelay$) and priority-based random testing ($RandomPriority$). These methods are practical to implement for large systems as they do not require static or dynamic analysis of the source code, but they build on a simple interception layer that can delay the delivery of a message. Delay-based testing method delays the delivery of the messages for a randomly selected amount of time. The priority-based testing method assigns random priorities to the messages and delivers them in the order of their priorities.

Second, we design an evolutionary search-based test case generation algorithm to investigate whether we can improve the performance of random testing by guiding the search for problematic executions. We describe two different fitness functions to direct the search toward certain system behaviors: $TimeFitness$ to direct the search to the executions that take longer time, and $ProposalFitness$ to direct the search to the executions that use more number of proposals to achieve an agreement. We evaluate the performance of the testing methods on three versions of the RCA source code that we seeded with some concurrency bugs.

Our experimental results show that random testing is effective at detecting concurrency bugs in large-scale blockchain systems, and our evolutionary search-based test case generation improves the testing performance. In our evaluation, the delay-based random concurrency testing method and the evolutionary testing approach could successfully detect all the concurrency bugs seeded in the system's code. Moreover, we discovered a previously unknown concurrency bug in the production implementation of RCA. The buggy execution violates termination, i.e., it causes the Ripple nodes to be stuck while trying to make an agreement for the set of transactions to be committed.

This paper makes the following contributions:

- A case study on the effectiveness of random concurrency testing on the Ripple blockchain system
- A new evolutionary algorithm for concurrency testing that guides the random generation of test cases toward certain behaviors of distributed systems
- Discovery of a new concurrency bug in the production implementation of Ripple
- A set of challenges and opportunities for future work on search-based concurrency testing

## II. Related Work

### A. Concurrency Testing for Distributed Systems

A large number of concurrency testing tools for distributed systems systematically explore the possible executions of a system, including dBug [5], MoDist [6], Concuerror [7], DeMeter [8], SAMC [9], and FlyMC [10]. These tools exhaustively exercise all possible reorderings of the concurrent events in execution, and they mainly differ in their target system under test. Although they employ partial order reduction techniques [11], [12] to reduce the set of generated test executions, they suffer from the state space explosion that makes them impractical to apply for large-scale systems and blockchains.

Alternative to systematic testing, randomized concurrency testing aims to detect buggy executions by running randomly generated orderings of the events in an execution. Jepsen [13], [14] and CoFI [15] test distributed systems by exercising their behaviors under random network partitions. Probabilistic concurrency testing (PCT) [16], [17] generate random test executions that provide nontrivial theoretical guarantees for detecting concurrency bugs. To increase the diversity of randomly generated test executions, RaPOS [18], TaPCT [19], and Morpheus [20] employ partial order reduction; the work in [21] exploits semantic reduction, and QL [22] uses reinforcement learning to achieve high coverage of system behaviors. While there has been some work on directing multithreaded program test executions towards increased coverage [23], [24], or guiding the exploration towards bug patterns in multithreaded programs [25], [26], they are not directly applicable to exploration of distributed consensus systems towards subtle concurrency behaviors.

Previous work showed that random testing outperformed systematic search at finding errors in real-world concurrent programs [4] and theoretically explained the effectiveness of random test generation [14] for covering a large set of behaviors correlated with distributed system bugs. In this work, we aim to investigate the effectiveness of random concurrency testing algorithms for detecting concurrency errors in the large-scale blockchain system, Ripple. Then, we explore whether we can improve the test efficiency by guiding the generation of test cases using an evolutionary algorithm.

### B. Test Case Generation using Evolutionary Algorithms

Evolutionary algorithms (EAs) have been widely used in the literature to automate the process of generating test cases. Within the search-based software testing (SBST) umbrella, EAs have been successfully applied to evolve and optimize test cases toward achieving desired testing goals [27]. Given a set of testing goals (e.g., branches in the code), EAs are guided by a fitness function that measures how distant the test execution is from reaching those goals [28]–[30]. The existing body of research has shown how EAs are particularly effective for different testing levels, including unit (e.g., [31]), integration [32], and system level [33], [34]. Furthermore,

EAs outperform random testing when the goal is to achieve high coverage [35], detect unit-level bugs [36], [37], or testing complex systems [34]. In this work, we aim to investigate the effectiveness of EAs when applied to detect concurrency bugs in Ripple's large-scale blockchain application.

## III. The Ripple Consensus Algorithm

### A. RCA Overview

The XRP Ledger of the Ripple blockchain uses the Ripple Consensus Algorithm (RCA)[1] to achieve agreement on the global order of blocks of transactions in the Ripple network. Unlike the early blockchain systems such as Bitcoin and Ethereum, RCA builds on the classical Byzantine fault-tolerance (BFT) style consensus and follows the basic design principles of the seminal Practical Byzantine Fault Tolerance (PBFT) algorithm [40]. Similar to PBFT, RCA is Byzantine fault-tolerant, i.e., it can tolerate malicious process behaviors, also known as Byzantine behavior [41], to a certain degree. Different from PBFT, which assumes a known set of protocol participants, Ripple is designed for the blockchain ecosystem that offers open membership. Each node in the Ripple network defines the set of protocol participants it trusts in a so-called Unique Node List (UNL) and runs the protocol using the votes from these nodes. RCA guarantees correctness as long as 80% of the UNL nodes are honest, i.e., non-Byzantine.

### B. RCA Protocol Steps

The Ripple consensus protocol runs as a sequence of synchronized consensus rounds in each of which the nodes in the network agree on a set of transactions to be executed. The Ripple nodes can receive transactions at any time concurrently to the execution of the consensus protocol rounds.

The design and implementation of RCA heavily rely on a common notion of time and synchronous execution of the nodes in the network. The transitions between the protocol steps are identified by predefined durations of time intervals, which are triggered by periodic timer messages.

Figure 1 illustrates an execution of the protocol with two client transactions that are submitted to the nodes $p1$ and $p2$, respectively. For simplicity, we show the message exchanges only between $p1$ and $p2$. We denote the protocol messages by rectangles and internally created objects with diamonds.

The execution of a consensus round has three phases: *open*, *establish*, and *accepted*.

*a) Open phase:* The nodes collect the submitted transactions to include in the next ledger and disseminate the received transactions to the other nodes (`TX` and `TX'` messages respectively, in Figure 1) in their UNLs. A node stays in the open phase for half of the duration of the previous consensus round and then moves to the establish phase.

[1]Ripple's consensus protocol is referred to as Ripple Consensus Algorithm (RCA) in the white paper [38] and as XRP Ledger Consensus Protocol (XRP LCP) in the updated presentation of the protocol [39].

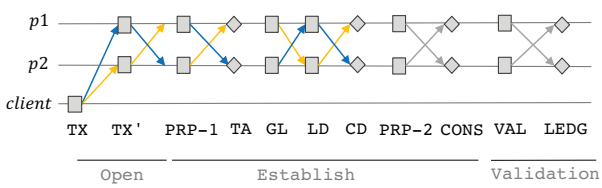



Fig. 1: An execution of the Ripple Consensus Algorithm (RCA). For simplicity, we show the message exchanges only between $p1$ and $p2$. We denote the protocol messages with rectangles and internally created objects with diamonds.

*b) Establish phase:* The nodes try to reach a consensus on the set of transactions to be included in the next ledger. They repeatedly exchange the set of transactions they propose for the next ledger in the proposal messages (`PRP` messages in Figure 1)). The sets of transactions proposed by the nodes can differ due to network asynchrony or process faults. A node receiving proposals from others creates a list of *disputed* transactions, which are the set of transactions not supported by the nodes in the UNL. These include transactions in a received proposal but not existent in their own proposal or transactions in their own proposal but not in the received proposal.

The nodes update their proposals by adding new transactions if most of the nodes in their UNL propose these transactions or removing disputed transactions from their proposal if most of the nodes in their UNL dispute them. The nodes iteratively send the updated proposal to the other nodes in its UNL. Transactions become disputed with an increasing threshold through an avalanche protocol. At the start of the establish phase, transactions become disputed when less than 50% of a node's UNL proposals contain it. This threshold increases to 65%, 70%, and finally 95% as the duration of the establish phase compared to the previous round's duration increases.

A node declares consensus on the transaction set if it reaches agreement with 80% of the nodes in its UNL before a predefined duration of time. If the node reaches consensus, it moves to the validation phase. Otherwise, it returns to the open phase.

*c) Validation phase:* The nodes validate that they decided on the same ledger and finalize the ledger version. They compute the ledger from the agreed set of transactions and send the new ledger's hash in a validation message (`VAL` in Figure 1). The nodes collect validations until they receive the same validated ledger hash from $\geq$80% of the nodes in their UNL. When the new ledger is then fully validated, the nodes apply transactions in the ledger, which are final and irreversible.

**Example.** In the example execution given in Figure 1, the client submits two conflicting transactions (e.g., each of which spends all the money in the same account) to different nodes $p1$ and $p2$ (`TX` messages). In the open phase, the nodes add the transactions they receive to their open ledgers. The node $p1$ adds the blue transaction, $p2$ adds the yellow transaction in their open ledgers, and they relay the transactions they

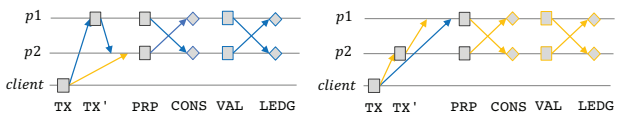



Fig. 2: Two alternative executions of RCA due to concurrency nondeterminism. The execution on the left commits the blue transaction, and the right commits the yellow transaction.

received (TX′ messages). The relayed transactions are discarded as they conflict with the transaction already in the open ledger. In the establish phase, the nodes share their proposed ledgers (PRP−1), and they discover that they are missing a transaction. They create a TransactionAcquire (TA) object, which periodically broadcasts GetLedger (GL) messages in an attempt to acquire the missing transaction. Upon receiving the GL message, the node with the requested transaction sends a LedgerData (LD) message containing the transaction. The nodes $p1$ and $p2$ acquire the missing transactions during the establish phase and they create disputes (CD). Since the transactions in this example conflict with each other (e.g., they carry the same sequence number), none of the transactions get a sufficient amount of votes, and both nodes remove their respective transactions. The nodes propose ledgers with an empty set of transactions (PRP−2), achieve consensus, and validate the empty ledger.

### C. Concurrency Nondeterminism in the Executions

Due to asynchronous message communication and the concurrency nondeterminism in the delivery of protocol messages, a set of client transactions can be processed by many possible executions. When conflicting transactions are submitted simultaneously to different nodes, the delivery order of concurrent messages determines the transactions that are committed.

The execution in Figure 1 commits none of the two conflicting transactions because the transaction submission messages (TX) are delivered before the transaction relay messages (TX′). Here, we provide two alternative executions, in one of which the network commits the blue transaction, and in the other, it commits the yellow transaction.

Figure 2 illustrates how different delivery orderings of messages result in different transactions being committed in a ledger. In the execution on the left, the blue transaction relay message from $p1$ to $p2$ (TX′) arrives before the yellow transaction submission message from the client to $p2$ (TX). Since $p2$ has already added the blue transaction in its open ledger, it refuses the conflicting yellow transaction from the client. The nodes agree on the ledger having the blue transaction and validate it. Alternatively, the yellow transaction's relay message can arrive before the blue transaction submission message from the client to $p2$ (as in the execution on the right). In that case, the yellow transaction is committed, and the blue transaction gets refused.

### D. Correctness Specification: Safety and Liveness Properties

The correctness of distributed consensus protocols is specified by the following *safety* and *liveness* properties [42]:

1) **Agreement** No two nodes decide differently.
2) **Validity** If a node decides a value, then that value was proposed by some node.
3) **Integrity** No node decides twice.
4) **Termination** Every node eventually decides some value.

In the context of blockchain consensus protocols, deciding a value refers to deciding on a ledger to be appended to the blockchain. The agreement, validity, and integrity are safety properties that ensure nothing *bad* will happen in the execution. An example of a safety violation is disagreement among the nodes that can cause forks in the network. The termination property is a liveness property that ensures something *good* will eventually happen. An example of a liveness violation is an execution where the nodes do not make any progress in processing client requests, rendering the system unresponsive.

## IV. Search-based Testing of RCA

The testing goal is to uncover concurrency bugs by searching over the space of possible schedules. A search algorithm has to be capable of changing schedules in a meaningful and direct way. Creating schedules before their execution runs the risk of creating infeasible schedules, e.g., an event might be scheduled at a moment when it is not enabled. Furthermore, due to the lack of control of the nodes' initial states, it is impossible to force the execution of a predetermined schedule. In practice, this means that the messages sent by the nodes in the network form an initial schedule. This schedule can be changed by reordering the delivery of messages. A challenge is that this initial schedule is non-deterministic and not known before executing the schedule. At this time, any search algorithm will need to be acting to change the schedule. Therefore, any changes to schedules need to be made online.

In this paper, we investigate three search algorithms: two variants of random search (RS) and an evolutionary algorithm (EA). In the following subsections, we describe how we customized these search algorithms for testing the Ripple consensus algorithm. While our paper focuses on this real-world and large-scale application, our approaches can be applied to test different consensus algorithms.

### A. Random Testing

Random testing (or random search) is the simplest search algorithm to implement [43], [44] and often recommended as a baseline to assess new testing techniques [45]. It naive randomly samples the search space of schedules and evaluates them to check whether a concurrency bug has been uncovered. In particular, the algorithm generates $N$ random schedules, runs them, and stores those revealing one or more bugs. We considered two different variants of random testing, which differ in how the test cases are *encoded* and executed. The following paragraphs detail the main characteristics of these two variants.

TABLE I: Message types mapped by the genotype

| Message Type | Description |
|---|---|
| ProposeSet-0 | Contains the transaction set that the sending node proposes for the next ledger. Monotonically increases with each subsequent proposal i. |
| ProposeSet-i | Contains the set of transactions for ProposeSeq = i |
| ProposeSetBowOut | Indicates that a node no longer actively participates in the current consensus round. |
| StatusChange | Indicates that a node closes an open ledger or accepts a new ledger. |
| Validation | Contains the ledger hash, and ledger sequence that a node believes should be validated. |
| Transaction | Contains a transaction submitted to the network. |
| HaveTransactionSet | Indicates that the sender node has acquired a particular transaction set. |
| GetLedger | Fetches transactions and ledgers from other nodes. |
| LedgerData | Sends transactions and ledgers to other nodes. |

*1) Priority-based Random Testing:* Priority-based random testing samples an ordering of messages in a distributed system execution using the basic Partial order Aware Concurrency Sampling (POS) algorithm [20], [46]. The algorithm randomly assigns a priority value for each message in flight and delivers them in the order of their priorities. Assignment of different priority values to the messages results in different delivery orderings of the messages.

A test case in priority-based random testing is represented as a sequence (or schedule) of events mapped to priorities, where each event is a tuple ⟨`sender`, `receiver`, `message`⟩ that represents the delivery of `message` from the `sender` node to the `receiver` node. The types of messages we use in a test case are listed in Table I, which are the messages exchanged in the execution of the Ripple consensus algorithm.

Each event in a test case is assigned a randomly generated priority value. The mapping between events and their priority is handled by a *priority scheduler*, whose pseudocode is shown in Algorithm 1. The scheduler collects the messages sent by the nodes in an *inbox* (line 3) and delivers them at a variable rate $r$. Each time the scheduler wants to execute an event, the event with the highest priority is picked from the inbox and executed (line 6). The inbox is implemented as a priority queue, sorted in descending order on the priority of the events.

Figure 4 shows an example execution of priority scheduling in a simplified execution of a network with two nodes and two message types. The event mapping is shown in Figure 3a. The vertical lines in Figure 4 depict the moments in time when a new event is picked from the inbox and executed. The events shown in the inbox are sorted on priority, where the highest in the column has the highest priority and is executed next.

This example changes the initial schedule

$$s_{init} = \langle p2 : PR\rangle, \langle p1 : PR\rangle, \langle p2 : SC\rangle, \langle p1 : SC\rangle,$$

to

$$s_{new} = \langle p1 : PR\rangle, \langle p2 : SC\rangle, \langle p1 : SC\rangle, \langle p2 : PR\rangle.$$

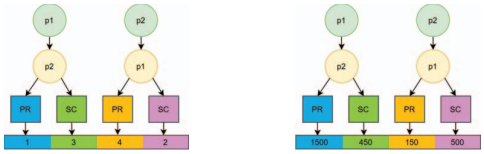
The variable rate at which the scheduler executes events is based on two objectives: (1) to have as many enabled events in the inbox as possible; and (2) to not delay events by too much. The first objective is to give the scheduler as many reordering options as possible. The second objective is not to delay events for too long, which will cause the receiver to ignore the message. The rate $r$ is the number of events executed per second. The base rate is equal to half the number of events $r_{base} = \frac{|events|}{2}$. Each time the scheduler executes an event, the rate is updated based on the inbox size (line 8 of Algorithm 1) as follows:

$$r_{i+1} = \begin{cases} \min\left(r_i \times s, |events|\right), & \text{if } |inbox| > \text{target} * \text{overflow}. \\ \max\left(\frac{r_i}{s}, \frac{|events|}{6}\right), & \text{if } |inbox| < \text{target} * \text{underflow}. \\ r_i, & \text{otherwise} \end{cases} \quad (1)$$

where $s$ is the sensitivity ratio; *target* denotes the target size

**Algorithm 1:** Pseudocode of the priority scheduler

```
Data: eventMapping, inbox, rate
/* A message is received from one of the nodes
   */
1  onRecvMessage(from, to, type):
       /* Get priority from eventMapping          */
2      priority ← eventMapping(from, to, type)
       /* Put into the inbox based on priority    */
3      inbox.push(Message, priority)
4  Function priorityScheduler():
5      Loop at rate
           /* Get the event with the highest
              priority                            */
6          message ← inbox.pop()
           /* Deliver the message                 */
7          execute(message)
           /* Adjust rate based on inbox size     */
8          rate ← adjustRate()
9      end
```




(a) Event mapping to priorities (b) Event mapping to delays

Fig. 3: Event mappings for (a) priority-based and (b) delay-based random scheduling.

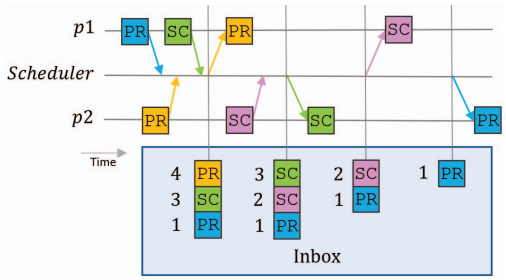



Fig. 4: An example execution of priority scheduling in a network with two nodes: $p1$ and $p2$, and two message types: `StatusChange` (SC) and `Proposal` (PR)

**Algorithm 2:** Pseudocode of the delay scheduler

```
   Data: eventMapping
1  Function delayScheduler():
       /* A message is received from a node      */
2      onRecvMessage(from, to, type):
           /* Get delay from eventMapping        */
3          delay ← eventMapping(from, to, type)
4          schedule(Message, delay)
5      end
6  Function schedule(Message, delay):
7      after(delay):          /* Wait for delay ms */
8          execute(Message)  /* Deliver the message */
9      end
```

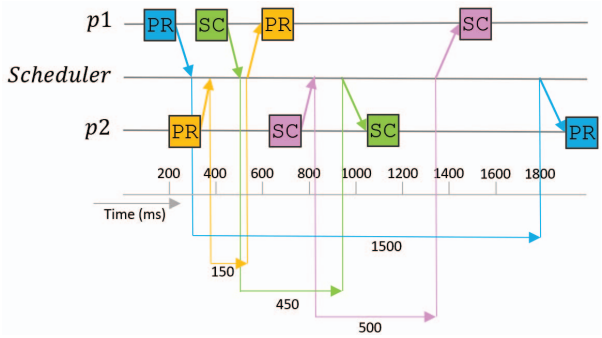



Fig. 5: An example execution of delay scheduling in a network with two nodes: $p1$ and $p2$, and two message types: `StatusChange` (SC) and `Proposal` (PR)

of the inbox; *overflow* is the percentage over the target inbox size; and *underflow* is the percentage under the target inbox size. The rate is clamped by $|events|/6 \leq r \leq |events|$.

*2) Delay-based Random Testing:* Delay-based random scheduling adopts early schedule perturbation methods for multithreaded concurrency [47], [48] to the distributed setting. Instead of delaying the execution of a thread event by inserting thread sleep statements, it delays the execution of a message by delaying its delivery. It delays the delivery of each message for a random amount of time and delivers them after that timeout.

This search algorithm employs a *delay scheduling*, which maps events to a time delay in milliseconds. Applying different delays to different events will reorder them and does not require collecting messages in an inbox. Hence, a test case is encoded as a vector of integers representing the time delay in milliseconds applied to each event. Figure 5 shows an example execution based on delay scheduling using the event mapping in Figure 3b.

Algorithm 2 shows the pseudocode for the delay scheduler. This scheduler does not have an inbox and rate. Instead, in `delayScheduler`, it continually listens to messages from nodes (`onRecv`). When a message is received, it looks up the delay in the event mapping and schedules the execution with `schedule`. This function waits for delay milliseconds before executing the message.

**Algorithm 3:** Pseudocode of the $(\mu + \lambda)$ EA

```
1  Function EA(μ, λ):
2      parents, offspring ← init(μ, λ)
3      while t < search_budget do
4          evaluate(offspring)
5          parents ← selection(parents + offspring)
6          offspring ← recombination(parents)
7      end
8
9  Function init(μ, λ):
10     initial_population ← sampleGenotypes(λ)
11     evaluate(initial_population)
12     parents ← selection(initial_population, μ)
13     offspring ← recombination(parents, λ)
14     return parents, offspring
15
16 Function recombination(parents, λ):
17     offspring ← crossover(parents, λ)
18     for individual ∈ offspring do
19         mutation(individual)
20     end
21     return offspring
```

### B. Evolutionary Testing

The third search algorithm we implemented and evaluated is the $(\mu + \lambda)$ evolutionary algorithm [49]. The algorithm (see Algorithm 3) starts with a population of $\lambda$ individuals. The best $\mu$ individuals are selected as parents to breed $\lambda$ offspring. The offspring individuals are created by recombining the parents solutions using the *crossover* and *mutation* operators as indicated in the `recombination` function in Algorithm 3. From the $\mu$ parents and $\lambda$ offspring, the best $\mu$ individuals are selected as parents for the next generation. This process repeats until the search budget is expended or a bug is found. The evaluation of one schedule takes roughly 20 seconds, so the values for $\mu$ and $\lambda$ are chosen to be rather small. $\mu = 4$ and $\lambda = 4$. Small values for $\mu$ and $\lambda$ are widely recommended in the literature for expensive fitness functions [45], [50].

In the following paragraphs, we describe the problem representation, the evolutionary operators, and the fitness functions used by the $(\mu + \lambda)$ evolutionary algorithm in detail.

*1) Encoding:* A solution (or individual) for the evolutionary algorithm is encoded (or represented) using the delay scheduling as done for the delayed-based random testing. Hence, a test case $T = [t_1, \ldots, t_n]$ is a vector of integers, where each element $t_i$ denotes the delay time (in millisecond) for the $i-th$ event in the test case. As done for the random testing, also in this case each event is a tuple $\langle$`sender node`, `receiver node`, `message type`$\rangle$, where the message type is one of the types in Table I.

*2) Crossover Operators:* The crossover operator recombines two individuals to create new individuals. We use a simulated binary crossover (SBX) [51], [52], which simulates one-point crossover in binary-encoded problems for real-valued vectors (as in our case). Two children $c_1$ and $c_2$ are created from two parents $p_1$ and $p_2$ as follows:

$$c_{1,k} = 0.5[p_{1,k} + p_{2,k} - \beta_k(p_{1,k} - p_{2,k})]$$
$$c_{2,k} = 0.5[p_{1,k} + p_{2,k} - \beta_k(p_{1,k} - p_{2,k})]$$

Where $c_{1,k}$ is the $k_{th}$ gene in $c_1$ and $p_{1,k}$, $p_{2,k}$ is the $k_{th}$ gene in $p_1$ and $p_2$ respectively. $\beta_k$ is a random number generated from the probability density function

$$p(\beta) = \begin{cases} 0.5(\eta_c + 1)\beta^{\eta_c}, & \text{if } 0 \leq \beta \leq 1 \\ 0.5(\eta_c + 1)\frac{1}{\beta^{\eta_c+2}}, & \text{if } \beta > 1 \end{cases}$$

$\eta_c$ is a user-chosen distribution index. According to the original SBX paper [52], values for $\eta_c$ between 2 and 5 closely match one-point crossover in binary-coded EAs. Smaller $\eta_c$ results in child genes further from the parent's genes and vice versa. We use $\eta_c = 3$. The individual genes are recombined with probability $0.5$, otherwise, the parent's genes are copied to the children. This probability is also used in [51].

*3) Mutation Operators:* The mutation operator changes individuals slightly to improve exploration. In Algorithm 3, the `mutation` function denotes when mutation is applied to an individual. We use the Gaussian mutation [53] for the mutation operator on delay genotypes. A gene $x_i$ is mutated by adding a sample from a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$, where $\mu = x_i$ and $\sigma = (b_i - a_i)/100$, $a_i \leq x_i \leq b_i$ [53]. The mutation probability is set to $\frac{1}{n}$, so that, on average, one gene gets mutated per input vector per generation. This probability is most commonly used in the literature [53].

*4) Fitness Function:* The effectiveness of evolutionary algorithms strongly depends on the guidance the fitness function provides. In our context, the guidance corresponds to the scores the fitness function assigns to different schedules and how it awards schedules closer to finding a concurrency bug.

Determining the proximity to finding a bug is difficult. Bugs come in many variations, each having different characteristics and symptoms. The defining characteristic of concurrency bugs is the cause: a specific interleaving of events, not the result. Therefore, measuring the proximity to a concurrency bug through a fitness function is not trivial. For this reason, we defined two *heuristics* (or fitness functions) that reward schedules resulting in rarer and more complex executions.

Given the space of schedules $S$, a fitness function

$$f : S \to \mathbb{R},$$

maps every schedule $s \in S$ to a real number, whose value depends on the result of the test case *TC* execution.

**Time fitness**. The first heuristic (or fitness function) measures the time taken to complete the test case, i.e., to complete all its events:

$$f_t(s) = TC(s)_{time} \tag{2}$$

Intuitively, as the nodes take longer to validate the submitted transactions, the schedule can result in a more complex execution. In addition, this fitness function directly rewards schedules closer to violating RCA's termination property.

**Proposal fitness**. This second fitness function utilizes the sequence number carried in proposal messages. As nodes have more difficulty reaching an agreement on the transaction set, they send more proposal messages in a single consensus round. Each subsequent proposal message from a node carries a higher sequence number. A fitness function that rewards schedules with higher maximum proposal sequences can guide the algorithm towards schedules that result in more complex establish phases and deliberation rounds.

A bowout proposal contains a sequence number of 4294967295 (The highest unsigned 32 bit number), which would by definition reward a $TC(s)$ with $e = \langle \_, \_, \texttt{prop.bowout} \rangle \in s$, the highest possible fitness value. This is undesirable, but we do want to use this information in the fitness function. Bowout proposals indicate a node switched ledgers during consensus, which does not frequently happen in common executions. Therefore, we add the number of bowout proposals in the schedule to the fitness function. To preserve the relative influence of both parts of the fitness function, we scale the highest proposal sequence number by the number of nodes $n$ in the network. The function below excludes bowout proposals from `prop` and denotes these as `bowout`:

$$f_p(s) = n \times \max_{e=\langle \_,\_,\texttt{prop} \rangle \in s} \texttt{prop.seq} + |\{e : e = \langle \_, \_, \texttt{bowout} \rangle \in s\}| \tag{3}$$

## V. Case Study on the Ripple XRP Ledger

Our case study on the Ripple XRP Ledger tests the Ripple source code to search for concurrency bugs in the RCA consensus implementation. It aims to evaluate:

I *The effectiveness of random concurrency testing methods on the production implementation of Ripple XRP Ledger*
II *The effectiveness of guiding random test case generation using evolutionary algorithms toward the executions with consensus violations*
III *The comparison of guiding random test case generation using time fitness and proposal fitness functions.*

We evaluate the performances of the concurrency testing methods by running them on three different versions of the Ripple source code that we seeded with bugs.

First, we tested each version using two random concurrency testing methods:

- $RandomDelay$: that delays the delivery of the in-flight messages for random amounts of time. The maximum delay in delay scheduling is 4000 ms. This delay allows the network to make progress while providing the scheduler with a large enough window to reorder events.
- $RandomPriority$: that assigns random priorities to the messages and executes them in their priority order.

Then, we applied our evolutionary test generation algorithm in Section IV-B to guide executions of $RandomDelay$ using either of the two fitness functions:

- $TimeFitness$: that directs the test cases towards the executions that take more time to complete.
- $ProposalFitness$: that directs the test cases towards schedules with higher maximum proposal sequences.

### A. Experimental Setup

We tested the Ripple software v.1.7.2[2] on a private peer-to-peer network. The network consists of five Ripple server nodes in docker containers, where each node has all other nodes in its UNL. We performed the experiments on virtual machines with 2 Intel Xeon vCPU's @2.6GHz and 4GB memory.

As the test harness, we concurrently submit the following four transactions to the Ripple network: $Tx_1 = \{1, 1, 2, 80\}$, $Tx_2 = \{2, 1, 3, 80\}$, $Tx_3 = \{3, 1, 3, 80\}$, $Tx_4 = \{4, 1, 2, 80\}$, where a transaction is represented by the transaction id, source account id, destination account id, and transfer amount.

The transactions operate on three XRP accounts. Account 1 has a starting balance of 80 XRP and attempts to spend its balance four times (twice to Account 2 and twice to Account 3). We submit the transactions to different Ripple server nodes simultaneously 2000 ms after the start of the test execution. A test case consists of submitting transactions to the network and waiting for these transactions to be validated.

We tested three versions of the Ripple source code, two of which we seeded bugs that expose only under some delivery orderings on the protocol messages:

- **B1 (with a seeded bug in the processing of proposal messages)**: A modified version of the Ripple source code in which the nodes do not check the monotonicity of the sequence number carried in proposal messages from other nodes. This allows an older proposal to override a more recent one, enabling nodes to declare consensus on different transaction sets more easily, which violates the safety property of the agreement.
- **B2 (with a seeded bug in the validation threshold values)**: A modified version of the Ripple source code in which the nodes validate proposals when they reach a quorum threshold of 40% agreement in the UNL instead of 80%. This can result in two nodes validating two different ledgers, which violates the safety property of the agreement.
- **B3 (the original source code)**: The unmodified version of the Ripple source code. Our tests detected a previously unknown bug that violates the liveness property of termination as explained in Section V-E.

### B. Correctness Specification

After running each test case, we check whether the execution adhered to the consensus properties given in Section III-D. We check the following properties for RCA:

**Termination:** A violation of a liveness property, such as termination, can be identified by an infinite execution that does not satisfy the property. In our evaluation, we check *bounded termination*, which is violated if execution takes more time than a predetermined upper bound.

We determined the upper bound on the maximum execution time for a test case based on the following parameters of RCA:

- *ledgerIDLE_INTERVAL = 15 sec*: The maximum duration a ledger may remain idle before closing
- *ledgerMAX_CONSENSUS = 10 sec*: The maximum duration to spend pausing for laggards
- *proposeFRESHNESS = 20 sec*: How long a proposal is considered fresh
- *validationFRESHNESS = 20 sec*: How long a validation is considered fresh

Summing up the parameters, we set the upper bound to 65 seconds. We mark an execution as a violation of bounded termination if it requires more time than that. Under normal circumstances, RCA validates a ledger every four or five seconds, which is significantly faster than our upper bound.

**Validity:** We mark an execution as a violation of validity if:

1) A node declares consensus on a transaction set containing a transaction that was never proposed by any node in that consensus round, or
2) A node sends a validation message for a ledger that was not constructed by any node, or
3) During ledger switching, a node switches to a ledger chain that is not supported by any node

**Integrity:** We mark an execution as a violation of integrity if:

1) In one consensus round, a node declares consensus on the transaction set twice, or
2) A node sends a validation message for a ledger with a sequence number for which it has already sent a validation.

**Agreement:** We mark an execution as a violation if:

1) Two nodes declare consensus on different transaction sets,
2) Two nodes validate two different ledgers.

### C. Methods used for Performance Analysis

For each version of the Ripple source code, we executed each search algorithm multiple times and recorded whether one of the specifications reported in Section V-B was violated and the running time needed to achieve such violation. Multiple runs allow us to account for the randomized nature of the search algorithms as suggested by the existing guidelines [43]. Then, we compared the alternative algorithms w.r.t. their *success rate*, i.e., the number of times (over $n$ runs) they uncover a concurrency bug. To statistically assess the significance of the differences (if any), we applied Fisher's exact test with a significance level $\alpha = 0.05$ [54]. We opted for this statistical test because it is non-parametric, and it is well-suited to test dichotomous distributions [54], i.e., whether a bug is detected or not in each independent run in our context. We complemented our analysis with the *odds ratio* [55] (OR) to measure the magnitude of the differences (*effect size*). OR=1 indicates that the two algorithms in the comparison have the same success rate. Instead, OR>1 indicates that the first algorithms in the comparison achieved a larger success rate, while OR<1 indicates the opposite.

To deeper understand the performances of the different testing approaches, we also analyze their capability in detecting concurrency bugs in as little search budget as possible. While the success rate measures the algorithm performance only at the end of the search budget, we also want to analyze how

[2]https://github.com/ripple/rippled/releases/tag/1.7.2

| | $RandomPriority$ | $RandomDelay$ |
|---|---|---|
| B1 | 1 | **10** |
| B2 | 0 | **3** |
| B3 | 0 | **7** |

TABLE II: The number of runs that discover bugs

| | $TimeFitness$ | $ProposalFitness$ | $RandomDelay$ |
|---|---|---|---|
| B1 | **30** | **30** | **30** |
| B2 | **21** | 17 | 10 |
| B3 | **23** | 16 | 20 |

TABLE III: The number of runs that discover bugs using different fitness functions compared to random search

quickly the algorithms assessed in this study can find each bug. To this aim, we compare the different algorithms w.r.t. the average time (among $n$ runs) needed to uncover each bug (*efficiency*). To statistically assess the efficiency results, we applied the Wilcoxon rank sum test with a significance level $\alpha = 0.05$ [54] and the Vargha-Delaney ($\hat{A}_{12}$). The former test is a non-parametric test to assess whether two algorithms in the comparison statistically differ in terms of efficiency (significance test). The Vargha-Delaney statistics complement the analysis by assessing the effect size. $\hat{A}_{12}$ >0.50 indicates that the first algorithm in the comparison is more efficient than the second one; $\hat{A}_{12}$ <0.50 means that the first algorithm in the comparison is less efficient than the second one; $\hat{A}_{12}$ =0.50 indicates that the two algorithms are equivalent.

For our analysis, we opted for non-parametric tests since they do not make any normality assumption for the distributions (i.e., success rate or efficiency) being compared.

### D. Experimental Results

*1) Effectiveness of random concurrency testing algorithms for detecting bugs in Ripple:* Table II shows the results of testing the three versions of the Ripple source code B1-B3 using two random concurrency testing methods: $RandomDelay$ and $RandomPriority$. We collected the test results for each of the algorithms using 10 runs, where each run tests the system for a duration of one hour. The table lists the number of runs out of 10 that detect bugs in the benchmarks.

In our evaluation, random concurrency testing using $RandomDelay$ outperformed $RandomPriority$ in detecting concurrency bugs. $RandomPriority$ detected the seeded bug in B1 in a single run and could not detect any bugs in the other benchmarks. On the other hand, $RandomDelay$ detected the seeded bugs in B1 and B2 in more runs. Moreover, it discovered a new bug that causes a violation of termination in Ripple's consensus. We explain the buggy execution scenario in detail in Section V-E.

*2) Effectiveness of guiding the test generation using the two fitness functions:* Table III shows the results of testing the three versions of the Ripple source code B1-B3, guiding the test generation with time fitness, proposal fitness, or without any guidance. We collected the test results for each of the algorithms using 30 runs, where each run tests the system for a duration of one hour. The table lists the number of runs out of 30 that detect bugs in the benchmarks.

Our evaluation shows that the evolutionary algorithm effectively guides the test generation toward buggy executions. The algorithm could detect both bugs seeded in benchmarks B1 and B2 and the new liveness bug in B3. We can explain the difference in the bug detection rates in B1 and B2 by the characteristics of the concurrency bugs seeded in these benchmarks. The bug in B1 violates the agreement property (1) reaching consensus on two different transaction sets. The bug in B2 violates the property of agreement (2) as it validates two different ledgers. Validating two different ledgers should only be possible when two nodes apply different transaction sets to their ledgers, making B2 a subset of B1, where breaking agreement(1) is made even easier due to the inserted bug.

**Success rate**

| | | Random | | Time | | Proposal | |
|---|---|---|---|---|---|---|---|
| | | $p$-value | OR | $p$-value | OR | $p$-value | OR |
| B2 | Time | 0.01* | 4.54 | - | - | 0.42 | 1,77 |
| | Proposal | 0.12 | 2.57 | 0.42 | 0.57 | - | - |
| B3 | Time | 0.57 | 1.63 | - | - | 0.10 | 2.82 |
| | Proposal | 0.43 | 0.58 | 0.10 | 0.35 | - | - |

TABLE IV: $p$-values and odds ratios of the success rate. A row contains the comparison of that row's configuration to each column's configuration. * indicates a statistically significant $p$-value

*3) Comparison of the two fitness functions:* We compare the performances of the two different versions of the evolutionary algorithm with different fitness functions against independently random testing by evaluating (1) the success rate of detecting the bug and (2) the time to detect the bug.

Tables IV and V show the success rate and efficiency of using each fitness function and random search, respectively. Since we found the concurrency bug in benchmark B1 in all runs for all configurations, we compare the performances for detecting bugs in B2 and B3.

We compare the performance of time fitness with proposal fitness and random search, and proposal fitness with time fitness and random search. We create a contingency table for each comparison and use Fisher's exact test to calculate the $p$-values and odds ratios.

Table IV shows the resulting $p$-values and odds ratios. The evolutionary test case generation using time fitness significantly outperforms random testing on benchmark B2 with $p$-value = $0.01 < \alpha$ and $OR = 4.54$. It performs no worse than independently random test generation on B1 and B3. Using proposal fitness, on the other hand, evolutionary test generation shows only marginal improvement in success rate over random testing on B2. However, the statistical significance analysis does not draw clear conclusions about the proposal fitness.

An observation from Table IV is the difference in success rates for the detection of the bugs in the benchmarks. While the evolutionary test case generation guided by the fitness functions outperforms random search (significantly with time

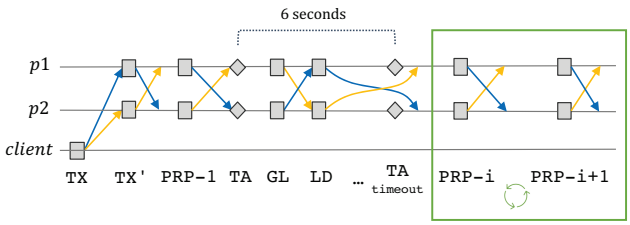



Fig. 6: The buggy execution triggering the liveness violation

fitness) for B2, this is not the case for B3. This can be explained by the characteristics of the concurrency bug in B3 (Section V-E), we detected in the tests that apply large delays on `GetLedger` and `LedgerData` messages. The bug is only triggered when the total delay of both message types is more than 5250 ms. The number of generations it takes for an evolutionary algorithm to reach the required delays largely depends on the initial population. If most individuals in the initial population have small delays for these message types, the evolutionary algorithm requires a higher number of generations to increase delays to the required values.

**Efficiency**

| | | Random | | Time | | Proposal | |
|---|---|---|---|---|---|---|---|
| | | $p$-value | $A_{12}$ | $p$-value | $A_{12}$ | $p$-value | $A_{12}$ |
| B1 | Time | 0.42 | 0.56 | - | - | 0.73 | 0.47 |
| | Proposal | 0.30 | 0.58 | 0.73 | 0.53 | - | - |
| B2 | Time | 0.49 | 0.55 | - | - | 0.48 | 0.57 |
| | Proposal | 0.67 | 0.55 | 0.48 | 0.43 | - | - |
| B3 | Time | 0.86 | 0.52 | - | - | 0.40 | 0.58 |
| | Proposal | 0.44 | 0.42 | 0.40 | 0.42 | - | - |

TABLE V: $p$-values and $A_{12}$ effect sizes of the efficiency. A row contains the comparison of that row's configuration to each column's configuration

We also compared the algorithms in terms of the time to reach the maximum test effectiveness [30], [56], which is the bug detection in our case. Table V shows the resulting $p$-values of the Wilcoxon tests and the $A_{12}$ statistics. Our results do not show any significant difference between the different algorithms' time to bug detection. This could be due to the search budget of one hour combined with the expensive fitness evaluations ($\approx$ 20 seconds). Each run only allows for approximately 180 evaluations, which could be improved by increasing the search budget.

### *E. A New Concurrency Bug Discovered in RCA*

In our case study, we discovered a previously unknown production bug during our evaluation using delay scheduling. The bug causes the nodes to get stuck in the establish phase of the protocol indefinitely, violating the termination of consensus. Our blackbox tests detected the buggy execution without any prior information about the implementation details of the system.

Figure 6 shows a simplified version of the buggy execution with two nodes and a client. The nodes start in the open phase. The client sends two conflicting transactions, $tx_1$ and $tx_2$, that attempt to double-spend. $tx_1$ (in blue) is sent to $p_1$ and $tx_2$ (in yellow) is sent to $p_2$. The nodes receive conflicting transactions from each other but do not include them in their open ledger as they have already included the other transaction. The nodes then proceed to the establish phase and send their proposals. On receipt of the proposals, they discover that they do not have the transaction included in them. In turn, they create a `TransactionAcquire` (TA) object, which periodically broadcasts `GetLedger` (GL) messages in an attempt to acquire the missing transaction. Nodes receiving the GL message that have the referenced transaction will reply with a `LedgerData` (LD) message containing the transaction. Normally, on receipt of the LD message, the transaction will be acquired, disputes can be created, and consensus proceeds as normal. However, a TA object lives only for 5250 ms, after which it times out.

The problem arises when the LD message for the transaction arrives after the TA object has timed out. The LD message is ignored, and the node cannot acquire the transaction. Subsequent proposals containing the transaction cannot be used to create disputes. The nodes resend their proposals every 12 seconds as a way to keep their proposal fresh but cannot make forward progress. The nodes are stuck indefinitely.

We have reported the bug to Ripple's development team, and it is currently under investigation.

## VI. Threats to Validity

*Internal validity*. The main potential threats to the internal validity are related to (1) using randomized algorithms and (2) measuring efficiency in terms of running time. To mitigate the first threat, we ran each randomized algorithm multiple times and drew our conclusions based on the overall distributions obtained across the different runs. We also applied sound statistical tests, namely the Fisher's exact test [54], the odds ration [55], the Wilcoxon rank-sum test [57], and the Vargha-Delaney ($\hat{A}_{12}$) statistics. Hence, we carefully followed the existing guidelines regarding assessing randomized algorithms.

To alleviate the second threat, we implemented all algorithms and fitness functions in the same tool. Besides, we run all algorithms on the same machine to avoid potential bias that can manifest when using different environments.

*External validity*. The testing approaches described in this paper have been applied to a single (albeit large-scale) consensus algorithm developed by a single company (Ripple). Therefore, our conclusions w.r.t to the performance of evolutionary testing and different fitness functions might be specific to Ripple's code. Further research is needed to validate our results for other consensus algorithms developed and designed by other (research or industrial) entities.

## VII. Conclusion and Future Work

In this paper, we explored the concurrency behaviors of the Ripple XRP Ledger and uncovered a previously unknown concurrency bug in the Ripple production code. Our case study shows that randomized concurrency testing methods

successfully discover real-world concurrency bugs in large-scale systems. We further designed an evolutionary search-based concurrency testing algorithm to guide the test generation toward problematic executions. We proposed two fitness functions that direct the test generation towards the executions that take a long time or a long sequence of proposals to make a decision. In our evaluation, the time-based fitness function performed better at detecting concurrency bugs.

Having demonstrated the applicability and effectiveness of evolutionary search-based test generation for concurrency exploration, our case study points out several research directions.

*The impact of the schedule representation.* Test case representations provide different search performances depending on their locality, redundancy, and scaling [58], [59]. We presented two representations for delay and priority-based test generation. Future work can evaluate a wider set of representations.

*The impact of variation operations.* The selection of variation operators and parameters affects the search performance. Future work can explore a wider set of operations.

*The impact of the fitness function.* Fitness functions drive the test execution toward specific system behavior, and hence different fitness functions can be useful for searching for different goals. We presented two fitness functions that direct test cases toward longer execution time or higher proposal sequences. While these functions successfully guided the search toward termination violations, future work can analyze other fitness functions targeted at different types of violations. Furthermore, these fitness functions can be considered in combination rather than as alternatives using multi- and many-objective optimization algorithms, e.g., MOEA/D [60], AGE-MOEA [61], [62], or MOSA [30], [34]. We could also consider resource usage [63] as an additional factor to take into account when generating test cases also for regression purposes.

*Application to other systems.* While this paper presents a case study on Ripple, the proposed method is applicable for testing the consensus implementations in other distributed systems and blockchains that use a BFT-style consensus mechanism. Future work can explore the performance of different representations and fitness functions for testing other systems.

## ACKNOWLEDGEMENTS

This work has been supported by the University Blockchain Research Initiative (UBRI) project, funded by Ripple. We also thank Ripple's developers for their feedback and for discussing the results of this work.

## REFERENCES

[1] F. S. Board, "Assessment of risks to financial stability from crypto-assets," Financial Stability Board, Tech. Rep., 2022.

[2] H. S. Gunawi, T. Do, A. Laksono, M. Hao, T. Leesatapornwongsa, J. F. Lukman, and R. O. Suminto, "What bugs live in the cloud?: A study of issues in scalable distributed systems," *login Usenix Mag.*, vol. 40, no. 4, 2015.

[3] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "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*, T. Conte and Y. Zhou, Eds. ACM, 2016, pp. 517–530.

[4] P. Thomson, A. F. Donaldson, and A. Betts, "Concurrency testing using controlled schedulers: An empirical study," *ACM Trans. Parallel Comput.*, vol. 2, no. 4, pp. 23:1–23:37, 2016.

[5] J. Simsa, R. Bryant, and G. A. Gibson, "dBug: Systematic evaluation of distributed systems," in *5th International Workshop on Systems Software Verification, SSV'10, Vancouver, BC, Canada, October 6-7, 2010*, R. Huuck, G. Klein, and B. Schlich, Eds. USENIX Association, 2010.

[6] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: transparent model checking of unmodified distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, J. Rexford and E. G. Sirer, Eds. USENIX Association, 2009, pp. 213–228.

[7] A. Gotovos, M. Christakis, and K. Sagonas, "Test-driven development of concurrent programs using concuerror," in *Proceedings of the 10th ACM SIGPLAN workshop on Erlang, Tokyo, Japan, September 23, 2011*, K. Rikitake and E. Stenman, Eds. ACM, 2011, pp. 51–61.

[8] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practical software model checking via dynamic interface reduction," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, T. Wobber and P. Druschel, Eds. ACM, 2011, pp. 265–278.

[9] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, J. Flinn and H. Levy, Eds. USENIX Association, 2014, pp. 399–414.

[10] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa, A. Gupta, S. Lu, and H. S. Gunawi, "FlyMC: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse, and C. Fetzer, Eds. ACM, 2019, pp. 20:1–20:16.

[11] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032.

[12] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds. ACM, 2005, pp. 110–121.

[13] K. Kingsbury., "Jepsen," http://jepsen.io/. [Online]. Available: http://jepsen.io/

[14] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?" *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 46:1–46:24, 2018.

[15] H. Chen, W. Dou, D. Wang, and F. Qin, "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, 2020, pp. 536–547.

[16] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "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*, J. C. Hoe and V. S. Adve, Eds. ACM, 2010, pp. 167–178.

[17] B. Kulahcioglu Ozkan, R. Majumdar, F. Niksic, M. T. Befrouei, and G. Weissenbacher, "Randomized testing of distributed systems with probabilistic guarantees," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 160:1–160:28, 2018.

[18] K. Sen, "Effective random testing of concurrent programs," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM, 2007, pp. 323–332.

[19] B. Kulahcioglu Ozkan, R. Majumdar, and S. Oraee, "Trace aware random testing for distributed systems," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 180:1–180:29, 2019.

[20] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, J. R. Larus, L. Ceze, and K. Strauss, Eds. ACM, 2020, pp. 1141–1156.

[21] C. Dragoi, C. Enea, B. Kulahcioglu Ozkan, R. Majumdar, and F. Niksic, "Testing consensus implementations using communication closure," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 210:1–210:29, 2020.

[22] S. Mukherjee, P. Deligiannis, A. Biswas, and A. Lal, "Learning-based controlled concurrency testing," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 230:1–230:31, 2020.

[23] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 221–230.

[24] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, G. T. Leavens and M. B. Dwyer, Eds. ACM, 2012, pp. 485–502.

[25] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds. ACM, 2008, pp. 11–21.

[26] S. Park, S. Lu, and Y. Zhou, "Ctrigger: exposing atomicity violation bugs from their hiding places," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, M. L. Soffa and M. J. Irwin, Eds. ACM, 2009, pp. 25–36.

[27] P. McMinn, "Search-based software testing: Past, present and future," *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 153–163, 2011.

[28] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, G. S. Avrunin and G. Rothermel, Eds. ACM, 2004, pp. 119–128.

[29] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[30] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 2015, pp. 1–10.

[31] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 2017, pp. 263–272.

[32] P. Derakhshanfar, X. Devroey, A. Panichella, A. Zaidman, and A. van Deursen, "Generating class-level integration tests using call site information," *IEEE Transactions on Software Engineering*, 2022.

[33] A. Arcuri, "Evomaster: Evolutionary multi-context automated system test generation," in *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 394–397.

[34] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Testing autonomous cars for feature interaction failures using many-objective search," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 143–154.

[35] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Inf. Softw. Technol.*, vol. 104, pp. 236–256, 2018.

[36] J. Campos, Y. Ge, N. M. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," *Inf. Softw. Technol.*, vol. 104, pp. 207–235, 2018.

[37] S. Panichella, A. Gambi, F. Zampetti, and V. Riccio, "SBST tool competition 2021," in *14th IEEE/ACM International Workshop on Search-Based Software Testing, SBST 2021, Madrid, Spain, May 31, 2021*. IEEE, 2021, pp. 20–27.

[38] D. Schwartz, N. Youngs, A. Britto *et al.*, "The Ripple Protocol Consensus Algorithm," *Ripple Labs Inc White Paper*.

[39] B. Chase and E. MacBrough, "Analysis of the XRP ledger consensus protocol," *CoRR*, vol. abs/1802.07242, 2018.

[40] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*, M. I. Seltzer and P. J. Leach, Eds. USENIX Association, 1999, pp. 173–186.

[41] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[42] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[43] A. Arcuri and L. C. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test. Verification Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.

[44] C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, and A. Panichella, "Single and multi-objective test cases prioritization for self-driving cars in virtual environments," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2022.

[45] R. B. Abdessalem, A. Panichella, S. Nejati, L. C. Briand, and T. Stifter, "Automated repair of feature interaction failures in automated driving systems," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 88–100.

[46] X. Yuan, J. Yang, and R. Gu, "Partial order aware concurrency sampling," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10982. Springer, 2018, pp. 317–335.

[47] S. D. Stoller, "Testing concurrent java programs using randomized scheduling," *Electron. Notes Theor. Comput. Sci.*, vol. 70, no. 4, pp. 142–157, 2002.

[48] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," *IBM Syst. J.*, vol. 41, no. 1, pp. 111–125, 2002.

[49] T. Weise, Y. Wu, R. Chiong, K. Tang, and J. Lässig, "Global versus local search: the impact of population sizes on evolutionary algorithm performance," *J. Glob. Optim.*, vol. 66, no. 3, pp. 511–534, 2016.

[50] T. Chugh, K. Sindhya, J. Hakanen, and K. Miettinen, "A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms," *Soft Comput.*, vol. 23, no. 9, pp. 3137–3166, 2019.

[51] K. Deb, "An efficient constraint handling method for genetic algorithms," *Computer Methods in Applied Mechanics and Engineering*, vol. 186, no. 2, pp. 311–338, 2000.

[52] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex Syst.*, vol. 9, no. 2, 1995.

[53] K. Deb and D. Deb, "Analysing mutation schemes for real-parameter genetic algorithms," *Int. J. Artif. Intell. Soft Comput.*, vol. 4, no. 1, pp. 1–28, 2014.

[54] P. Sprent, *Fisher Exact Test*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 524–525.

[55] S. E. Ahmed, *Effect Sizes for Research: A Broad Application Approach*, 2006, vol. 48, no. 4.

[56] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 122–158, 2018.

[57] W. J. Conover, *Practical Nonparametric Statistics*. Wiley, 1998.

[58] F. Rothlauf, *Representations for Genetic and Evolutionary Algorithms*. Springer Berlin, Heidelberg, 2006.

[59] ——, *Design of Modern Heuristics*, 1st ed., ser. Natural Computing Series. Springer Berlin, Heidelberg, 2011.

[60] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on evolutionary computation*, vol. 11, no. 6, pp. 712–731, 2007.

[61] A. Panichella, "An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization," in *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, A. Auger and T. Stützle, Eds. ACM, 2019, pp. 595–603.

[62] ——, "An improved pareto front modeling algorithm for large-scale many-objective optimization," in *GECCO '22: Genetic and Evolutionary Computation Conference, Boston, Massachusetts, USA, July 9 - 13, 2022*, J. E. Fieldsend and M. Wagner, Eds. ACM, 2022, pp. 565–573.

[63] G. Grano, C. Laaber, A. Panichella, and S. Panichella, "Testing with fewer resources: An adaptive approach to performance-aware test case generation," *CoRR*, vol. abs/1907.08578, 2019.