Statistical Bug Isolation for Consensus Systems

Levin N. Winter[1]

Supervisor(s): Burcu Kulahcioglu Ozkan[1]**, Ege Berkay Gülcan**[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 25, 2023

Name of the student: Levin N. Winter
Final project course: CSE3000 Research Project
Thesis committee: Burcu Kulahcioglu Ozkan, Ege Berkay Gülcan, Johan Pouwelse

Abstract

The testing of consensus systems has received growing attention and recent testing tools generate many faulty executions. However, there is a lack of methods that automatically analyze these outputs to identify the root causes of the bugs they found.

This paper presents ISOLATION, a statistical bug isolation algorithm that uses message-based predicates to discriminate between different faults. We applied our method to executions of the XRP Ledger blockchain and evaluated its performance. Our comparison shows that ISOLATION correctly separates bugs by their root cause and consistently outperforms the state-of-the-art with higher F-scores.

1 Introduction

The security of consensus algorithms is becoming increasingly important as blockchain applications rise in popularity. They are pivotal to the guarantees that blockchains make and are trusted financial assets by individuals worldwide. Bugs in said consensus algorithms can have catastrophic effects on the funds of all participants, hence, utmost care is taken to ensure the correctness of these systems [2].

In recent work, automated testing tools for implementations of consensus algorithms have been proposed [1; 3; 7; 11; 17; 23; 27] which explore different executions of the system under test to detect bugs. They aim to find scenarios in which the guarantees of the given consensus algorithm do not hold anymore, e.g. by violating a safety or liveness property. The testing tools use oracles that continuously monitor a given execution and check said properties to determine whether a run was successful or not. When a bug is found, the developer is presented with the inputs of the algorithm, a log of the system's execution, and an overview of the violated properties.

However, finding buggy executions is only the first step in the process of testing consensus algorithm implementations. To fix a bug its root cause must be identified. This task is often carried out manually by inspecting the sequence of messages exchanged between the processes and requires the developer to have a thorough understanding of the system's inner workings. While root causing a fault is already difficult for simple programs, the nondeterminism and concurrency that is inherent to consensus algorithms significantly adds to the complexity of the operation.

In the past, different algorithms have been proposed that support the process of root causing [10]. When executing them on a set of successful and failed program runs, they isolate a set of conditions that correlate with buggy behavior. These methods e.g. the field of statistical debugging [18] or data mining [26] have been shown to be effective on ordinary programs and simple message protocols.

These methods do not directly apply to isolating bugs in the executions of consensus systems, which is difficult due to the vast concurrency and nondeterminism they exhibit. That is because they either do not exploit properties intrinsic to consensus algorithms or fail to scale to more sophisticated protocols with complex bug triggering conditions.

To address this, we propose a new bug isolation algorithm, called ISOLATION, that is specific to consensus algorithms. It coarsens the existing model with additional insights of consensus systems to augment traces which the testing tools generate. To motivate and evaluate this research, we will answer the following research questions.

1. How well do existing methods identify bugs in consensus systems?
2. Which predicates effectively identify bugs in consensus systems?
   (a) Are multiple bugs correctly discriminated?
   (b) Do the predicates help to identify the underlying bug?

First, we motivate and introduce the ISOLATION algorithm in Section 3. We then describe our method in Section 4. The empirical evaluation in Section 5 begins by discussing the results of applying the existing method to traces of the XRP Ledger [6] which is subsequently compared to the results of ISOLATION.

2 Background

In this section, we provide the relevant background information for our ISOLATION algorithm. Section 2.1 gives a brief introduction to consensus algorithms and Section 2.2 describes the XRP Ledger. In Section 2.3, we explain what statistical bug isolation is.

2.1 Consensus Algorithms

Consensus algorithms play an important role in the field of distributed systems and are essential for ensuring agreement between a set of decentralized processes. Consensus systems are resistant to a limited number of crash or Byzantine, i.e. malicious, faults, depending on the specific protocol.

These systems ensure consensus by running several rounds of communication during which the participating processes exchange messages. These messages contain the type of the message, often referred to as the message's *verb*, and a body, containing the contents of the message. During each communication round, a process handles the messages it received in the previous round, updates its local state, and sends new messages to its peers.

The exact rules of communication depend on the particular protocol and, in the past, many different protocols have been proposed. While they each have their distinct guarantees, advantages, and disadvantages, they all ensure the correct replication of state in a distributed environment.

2.2 XRP Ledger

The XRP Ledger [6; 25] is a blockchain that enables global payments to settle within seconds. A network of $5f + 1$ of its validators is resilient to up to f Byzantine failures. This is achieved using a subjective trust assumptions for which each process maintains a list of other trusted processes, the so-called Unique Node List (UNL). During voting, a process only considers the votes of its trusted peers.

Figure 1 shows the normal execution of the XRP Ledger for two different ledgers i and $i + 1$. In our example, p_3

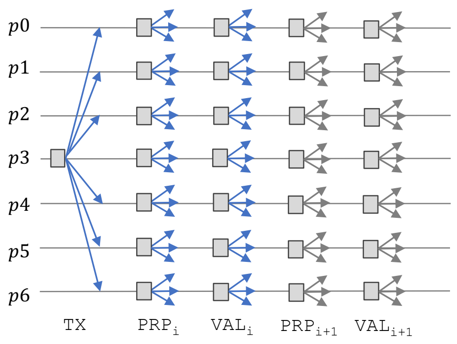

Figure 1: The normal case execution of the XRP Ledger. The image is taken from our recent work [27].

receives a transaction and disseminates it to its peers (blue arrows, `TX`). Subsequently, the processes enter into one or more `PROPOSAL` rounds, during which they establish consensus on the set of transactions to include in the next ledger. Figure 1 depicts a single round during which all processes propose the previously received transaction (blue arrows, $\texttt{PRP}_\texttt{i}$). If a process sees an agreement of less than 80%, it will update its vote and repeat the procedure. Once a quorum has been reached or a timer runs out, the processes move to the `VALIDATION` round. Here, a single `VALIDATION` message is sent, confirming the intent to validate the proposed ledger i (blue arrows, $\texttt{VAL}_\texttt{i}$). If a process receives matching `VALIDATION`s from at least 80% of its trusted peers, it fully validates the ledger and permanently commits its transactions. Since no new transactions were submitted before the following round, the processes first propose an empty ledger (gray arrows, $\texttt{PRP}_\texttt{i+1}$) and subsequently validate it (gray arrows, $\texttt{VAL}_\texttt{i+1}$).

2.3 Statistical Bug Isolation

Statistical bug isolation, a method from the field of statistical debugging, uses runtime data to identify certain predicates that correlate with different bugs. It relies on statistics and machine learning to isolate these bug-predicting conditions. When applying such a method to a data set containing both successful and failing executions of a program, the output can aid the developer in understanding the underlying root causes of the different bugs.

3 The ISOLATION Algorithm

In this section, we present ISOLATION, our novel bug isolation algorithm for consensus systems. In Section 3.1 we first discuss four key concepts that motivate our approach. Then, we introduce our predicates and formally describe them in Sections 3.2 and 3.3. Lastly, Section 3.4 outlines the underlying statistical framework.

3.1 Overview

ISOLATION is motivated by the four key ideas that (i) consensus algorithms communicate using message passing, (ii) they are stateful, (iii) a quorum is required to decide upon a client request, and (iv) they are resilient to a fraction of faulty processes. In the following paragraphs, we will provide some

$$i \times \texttt{VERB}_\text{L} \xrightarrow[\substack{\text{time}(l) < \text{time}(r) \\ \text{senders}(l) \subseteq \text{senders}(r)}]{\text{for more than } f \text{ processes}} \texttt{VERB}_\text{R}$$

Figure 2: Example of a predicate used in ISOLATION.

background on each of these ideas and explain their importance for ISOLATION.

Message Passing. Consensus systems mostly rely on message passing for communication because they are distributed systems for which shared-memory models are impractical [9]. The messages they exchange and their interleaving largely captures the state of the system. We therefore hypothesize that message-based predicates are sufficient to discriminate between consensus bugs. Because the ordering of messages from different senders is important, ISOLATION's predicates respect happened-before relations [14].

Stateful Systems. All consensus algorithms require a sequence of communication rounds to decide on a value. During each round, they process the newly received messages, update their local state, and send zero or more messages to other processes. The decisions taken in one round depend on the messages sent and received in the earlier rounds [5; 15; 22]. Thus, we designed the ISOLATION algorithm to also aggregate the messages received during a round and use the collected information during the evaluation of a predicate.

Quorums and Thresholds. To guarantee that the system agrees on a single value and no conflicting decisions are made, a quorum must be formed. Usually, a process determines that a quorum has been reached when it received a confirmation message from a certain number of other processes. Besides the quorum, there are other thresholds that, when being exceeded, affect the behavior of the process, e.g. because it changes its vote. ISOLATION's predicates capture these thresholds by requiring the receipt of i equal messages, which could cause the process to cross a threshold, before the receipt of another message.

Fault Tolerance. Consensus systems are resilient to a certain number of processes either crashing or diverging arbitrarily from the protocol, depending on the specific algorithm. The number of failures a system can tolerate is denoted by f and usually depends on the total number of processes n. If a bug only manifests in at most f processes, it might not impact the system. We model this in the ISOLATION algorithm by annotating the predicates with a tolerance. Only if the number of processes for which the predicate holds exceeds the threshold, we consider the predicate to be true.

3.2 Predicates

The aim of this work is to isolate different bugs in the set of faulty executions. For this, ISOLATION finds predictors that correlate with certain bugs. These predicates can help the developer to identify the underlying root cause of the bug.

Figure 2 shows an example of our predicates. Here, $\texttt{VERB}_\text{L}$ and $\texttt{VERB}_\text{R}$ are placeholders for two, possibly equal, message

```
Input: $n$, the number of processes
Input: $verbs$, the set of message verbs
Input: $a_{max}$, the maximum number of assertions per predicate
1  Procedure GeneratePredicates:
2      $predicates \leftarrow \emptyset$
3      for $(tolerance, threshold, a) \in [1,\dots,n] \times [1,\dots,n] \times [0,\dots,a_{max}]$ do
4          for $(v_l, v_r) \in verbs \times verbs$ do
               // find fields that can be compared
5              $comparable \leftarrow \emptyset$
6              for $(f_l, f_r) \in \text{fields}(v_l) \times \text{fields}(v_r)$ do
7                  if $\text{comparisons}(f_l, f_r) \neq \emptyset$ then   // a comparison operator exists
8                      $comparable \leftarrow comparable \cup \{(f_l, f_r)\}$
9              for $((f_{l_1}, f_{r_1}), \dots, (f_{l_a}, f_{r_a})) \in \text{subsetsOfSize}(comparable, a)$ do
                   // for each combination of comparison operators
10                 for $(*_1, \dots, *_a) \in \text{comparisons}(f_{l_1}, f_{r_1}) \times \dots \times \text{comparisons}(f_{l_a}, f_{r_a})$ do
11                     $assertions \leftarrow \{(f_{l_1}, *_1, f_{r_1}), \dots, (f_{l_a}, *_a, f_{r_a})\}$
12                     $predicates \leftarrow predicates \cup \{(tolerance, threshold, v_l, v_r, assertions)\}$
13     return $predicates$
```

Algorithm 1: Exhaustive generation of predicates which are input to the ISOLATION algorithm.

verbs. The arrow denotes the happened-before relation between the left and right side and i is the predicate's threshold. Above the arrow, we denote its tolerance f and a list of assertions that must all hold. We use them to capture complex relations between the bodies of the messages. The variables l and r represent the message bodies of $\texttt{VERB}_\text{L}$ and $\texttt{VERB}_\text{R}$.

For a single process, a predicate P can evaluate the three following possible results: *observed*, *observed to be true*, or *not observed*. The predicate is said to be *observed* if the process received messages of $\texttt{VERB}_\text{L}$, all having the same body, from at least i different senders followed by at least one occurrence of $\texttt{VERB}_\text{R}$. The messages are allowed to be interleaved with other messages, as long as the left-hand side happened-before the right-hand side. Otherwise, P is *not observed*. If a predicate is observed and all of its assertions hold, it can additionally be *observed to be true*.

We define the procedures $\text{observed}(P, m_l, s_l, m_r, s_r)$ and $\text{observedTrue}(P, m_l, s_l, m_r, s_r)$ which evaluate a predicate for a single process according to the aforementioned rules. They take as input the predicate P, the message m_l that corresponds to $\texttt{VERB}_\text{L}$ and its senders s_l, as well as the message m_r that corresponds to $\texttt{VERB}_\text{R}$ and its senders s_r.

Because an execution of a consensus system involves more than one process, we define how to evaluate a predicate against the full execution, as opposed to the previous definition for a single process. P is observed to be true for a run if the number of processes for which it is observed to be true exceeds P's tolerance f. If it is observed for at least one process, it is observed for the entire run. Otherwise, it is not observed. In the following, we use $\text{tolerance}(P)$ to refer to P's tolerance.

3.3 Pseudocode

We now describe how we generate the predicates for a consensus algorithm under analysis and evaluate them.

Generating Predicates

We present Algorithm 1 to generate the set of all predicates. For this, the list of all message verbs $verbs$, the number of processes in the consensus system n, and an upper bound on the number of assertions per predicate a_{max} must be supplied. Further, the algorithm relies on the following three subroutines. For a given verb, $\text{fields}(verb)$ returns a set of all fields that the verb's body contains. $\text{comparisons}(f_l, f_r)$ evaluates to the set of binary operators which can compare the two fields. All subsets of a given set that have a cardinality of a are returned by $\text{subsetsOfSize}(\{\dots\}, a)$.

The algorithm exhaustively generates all predicates using a number of Cartesian products and nested loops. For all combinations of the possible tolerances, thresholds, and amounts of assertions a, the following procedure is repeated (ln. 3). Each pair of message verbs is considered (ln. 4) and their pairs of comparable fields are determined (ln. 7). Then, for every subset of the matching fields that has size a (ln. 9), predicates are generated. This is done by considering each unique way in which the fields can be compared (ln. 10) and adding a new predicate for each of these.

The parameters n, the size of verbs $verbs$, and a_{max}, as well as the number of message fields, the number of compatible other fields, and the amount of operators that can compare them, all determine the number of possible predicates. An effective way to ensure the number of predicates does not explode is to keep the domains of the different parameters small and only compare message fields of the same datatype if they share similar semantics.

Evaluating Predicates

Algorithm 2 is used to evaluate a predicate P for a given execution of the consensus system. Besides P, the procedure takes the number of processes n as input. We use a set of helper functions where $\text{inbox}(i)$ returns an ordered list of all messages the specified process received. $\text{messages}(seen)$ extracts all messages from the $seen$ set and ignores their senders, whereas $\text{senders}(m, seen)$ returns the set of senders it has previously observed to send the given message m. For the definitions of $\text{tolerance}(P)$, $\text{observed}(P, m', s', m, s)$ and $\text{observedTrue}(P, m', s', m, s)$ we refer to Section 3.2.

For each process, the algorithm determines whether the

```
   Input: $P$, the predicate to evaluate
   Input: $n$, the number of processes
   Input: $\text{inbox}(i)$, the messages process $i$ received
1  Procedure EvaluatePredicate:
2      $observed \leftarrow$ false
3      $observed_{true} \leftarrow \emptyset$
4      for $i \leftarrow 1$ to $n$ do
5          $seen \leftarrow \emptyset$
6          for $(m, sender) \in \text{inbox}(i)$ do
7              for $m' \in \text{messages}(seen)$ do
8                  $s \leftarrow \text{senders}(m, seen) \cup \{sender\}$
9                  $s' \leftarrow \text{senders}(m', seen)$
10                 if $\text{observed}(P, m', s', m, s)$ then
11                     $observed \leftarrow$ true
12                 if $\text{observedTrue}(P, m', s', m, s)$ then
13                     $observed_{true} \leftarrow observed_{true} \cup \{i\}$
14             $seen \leftarrow seen \cup \{(m, sender)\}$
15     if $\text{tolerance}(P) < |observed_{true}|$ then
16         return $P$ is observed to be true
17     if $observed$ then
18         return $P$ is observed
19     return $P$ is not observed
```

Algorithm 2: Evaluation of a given predicate P for a run.

predicate P was not observed, observed, or observed to be true. This is achieved by iterating over the messages in a process' inbox in the order in which they were received (ln. 6). Every message m is compared to each previously received message m' (ln. 7). The senders of m' are directly determined from the $seen$ set, whereas for m we augment this information with the new sender of m (lns. 8, 9). If P is observed, we update the $observed$ variable, which is shared for all processes, to true (ln. 11). If P is observed to be true, we add the process' identifier to the $observed_{true}$ set (ln. 13). Finally, we add m and its sender to the $seen$ set (ln. 14).

If the number of processes that observed P to be true exceeds P's tolerance, P is observed to be true for the entire run (ln. 15). If at least one process observed P, P is said to be observed in the run (ln. 17). Otherwise, P is not observed in the run (ln. 19).

3.4 Statistics

The last step of our algorithm is the isolation of the most important predicates. For this, we use a simplified version of the statistical framework proposed in *Scalable Statistical Bug Isolation* [18]. In their work, they use code-based predicates that are randomly sampled throughout a program's execution. In contrast, ISOLATION evaluates every predicate for every run, allowing us to elide the parts related to the uncertainty introduced by the sampling. In the following, let $\mathrm{F}(P)$ be the number of failing runs in which P is observed to be true. We will now briefly summarize the most important steps.

To determine whether a predicate has predictive power, we calculate how much the probability of a failure increases when P is observed to be true compared to when P is simply observed. If the increase score is positive, it means that a fault becomes more likely when P is observed to be true. All predicates that have a negative increase are disregarded.

In a second step, the predicate with the highest importance is identified. For this, both the predicate's sensitivity, i.e. fraction of faults retrieved over total number of faults $numF$, and specificity, in this case the increase score, are combined using the harmonic mean. The score is computed as follows.

$$\text{Importance}(P) = \frac{2}{\frac{1}{\text{Increase}(P)} + \frac{1}{\log(\mathrm{F}(P))/\log(numF)}}$$

After returning the most important predicate to the developer, all runs for which this predicate was observed to be true are removed from the data set. This simulates how a human would fix the bug and hence remove all of its occurrences. The procedure is then applied recursively until all failing runs are clustered.

4 Method

This section outlines our method for the empirical evaluation. Section 4.1 describes how we generated the benchmark data set and Section 4.2 explains the baseline we selected. In Section 4.3 we state the metrics our evaluation utilizes and Section 4.4 gives details on the implementation.

4.1 Benchmark Data Set

To ensure a fair comparison between ISOLATION and the existing baseline algorithm, a common benchmark data set was generated. For this, we used our recent testing algorithm BYZZFUZZ [27]. By randomly mutating or dropping the messages sent between the processes, it introduces process and network faults into the consensus system's runtime to simulate Byzantine behavior. Compared to other testing algorithms, which e.g. only explore message reorderings [4], network partitions [13], or mimic loss of local state [3], BYZZFUZZ generates traces that have more diverse message contents, making it well suited for the evaluation of ISOLATION.

We chose the XRP Ledger for this benchmark because it is a widely used blockchain system. Recent works have exposed several bugs which we can use to measure ISOLATION's performance. For this, we have reintroduced a previously fixed vulnerability [27] and use a UNL configuration that is known to have insufficient overlap [2; 6].

4.2 Baseline Algorithm

For the baseline, we selected the method proposed by Libit et al. [18] as it has been shown to effectively isolate bugs in complex software systems. In our evaluation, we use a slightly modified version that only instruments branches and not scalar pairs and return values. This is supported by a quantitative analysis which indicated that there are relatively little scalar variables, because most of the logic is encapsulated in objects. Further, we hypothesize that the nondeterministic behavior is mostly determined by the branches and hence sufficiently captures the observable state.

4.3 Metrics

The two most important metrics to answer our research questions are a predicate's precision and recall. Here, the precision describes the percentage of runs that contain a specific bug, out of all of the runs associated with a predicate. It quantifies the discrimination ability of a given predicate.

Table 1: Distribution of bugs in the benchmark data set.

Bug	**#Runs**	**Mixed**
Correct Execution	1055	—
Incompatible Ledger	51	22
Insufficient Support	116	23
Agreement Violation	2	2
Total	1200	23

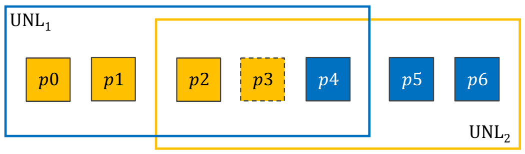

Figure 3: Schematic representation of the UNL setup.

However, it is not sufficient for a full evaluation. Every predicate that retrieves exactly one faulty run has a precision of 100.0% for at least one bug category. Thus, we additionally consider a predicate's recall, i.e. the number of bugs that were retrieved over the total number of bugs in that category. Together, these scores measure a predicate's performance.

F-scores are a standard way to combine both precision and recall into a single value. Throughout the evaluation, we report both the unbiased F$_1$ score, as well as the F$_{0.5}$ score that favors precision over recall. This is because having two similar predicates that split one bug into two sub-predictors is more favorable than having one predicate that combines two unrelated bugs.

4.4 Implementation

To implement the baseline algorithm, we developed a source-to-source compiler that automatically instruments the XRP Ledger's code. First, it uses the Clang compiler front-end [16] to parse the source code into an abstract syntax tree (AST). Then it traverses the AST to identify the locations of all if statements, while loops, and their respective conditions. Lastly, the source code is manipulated to insert logging statements at these locations to track the decisions taken.

The implementation of ISOLATION closely resembles Algorithms 1 and 2. We slightly modified the method to only consider the consensus-relevant `PROPOSAL` and `VALIDATION` messages as well as evaluate the predicates for each UNL separately. Both methods share a single implementation of the statistical framework.

5 Empirical Evaluation

This section presents the results of our empirical evaluation. We will first introduce the different bugs in our benchmark data set in Section 5.1. Following that, we explain the findings of the baseline method and our ISOLATION algorithm in Sections 5.2 and 5.3, respectively.

5.1 Benchmark Bugs

We generate the benchmark data set of executions by testing the XRP Ledger using the BYZZFUZZ testing algorithm, as

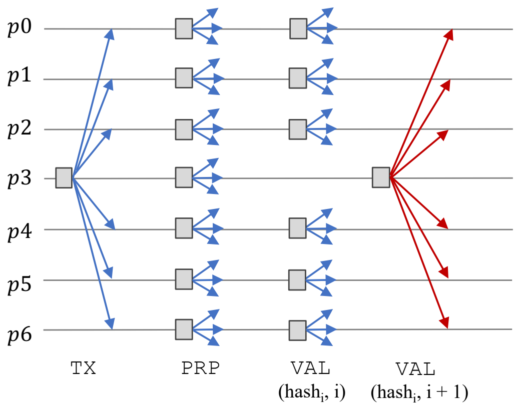

Figure 4: Example of the incompatible ledger bug.

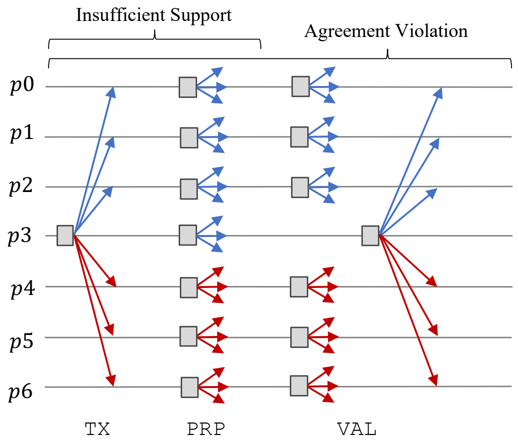

Figure 5: Example of insufficient support and agreement bug.

described in Section 4.1. The private XRP test network consists of seven so-called validators. The system makes use of two different UNLs: UNL$_1$ includes p_0 to p_4, whereas UNL$_2$ comprises p_2 to p_6. The first four processes trust UNL$_1$and the last three processes trust UNL$_2$, as shown in Figure 3.

Table 1 outlines the distribution of the three bug types in the data set. While a majority of the 145 faulty runs had a single root cause, 23 of them failed because of a combination of different errors. For each bug, we report the number of runs which include that fault in the *#Runs* column. Additionally, the *Mixed* column contains the number of runs which contained the indicated bug as well as at least one other bug.

We continue to briefly describe the three bugs. For a more detailed description, we refer to our BYZZFUZZ paper [27].

Incompatible Ledger

The *incompatible ledger* bug, recently discovered by our BYZZFUZZ testing algorithm and visualized in Figure 4, exposes due to the mishandling of certain Byzantine, i.e. malicious, validation messages. In the XRP Ledger, each process maintains a record of the hash and sequence number of the last fully validated ledger. When a process has fully validated a ledger with sequence i and hash $hash_i$, the pair $(hash_i, i)$

Table 2: The results of applying the scalable statistical bug isolation method to the XRP Ledger.

Predicate	#Runs	Bug	Precision	Recall	$\mathbf{F_1}$	$\mathbf{F_{0.5}}$
`if (seq != (ledger.seq() - 1))` `hashOfSeq(ledger, seq, journal)` `ledger/impl/View.cpp:655`	142	Incompatible	34.5%	96.1%	50.8%	39.6%
		Insufficient	81.0%	99.1%	89.1%	84.1%
		Agreement	1.4%	100.0%	2.8%	1.8%
`if (newPeerProp.prevLedger() != prevLedgerID_)` `peerProposalInternal(now, newPeerPos)` `consensus/Consensus.h:732`	2	Incompatible	50.0%	2.0%	3.8%	8.5%
		Insufficient	50.0%	0.9%	1.7%	4.0%
		Agreement	0.0%	0.0%	0.0%	0.0%
`if (!prevAgree || (seq % Seq{interval} != Seq{0}))` `getNextLedgerTimeResolution(prevRes, prevAgree, seq)` `consensus/LedgerTiming.h:110`	1	Incompatible	100.0%	2.0%	3.8%	9.1%
		Insufficient	0.0%	0.0%	0.0%	0.0%
		Agreement	0.0%	0.0%	0.0%	0.0%

is stored in an internal variable. If it now receives a validation for $hash_i$ but sequence j, where $i < j$, it will incorrectly overwrite the internal variable to $(hash_i, j)$, containing the hash of the ledger with a mismatched sequence number.

From now on, internal consistency checks fail when attempting to validate because the new ledger $hash_k$ with sequence k is not a valid descendant of, i.e. compatible with, the pair $(hash_i, j)$. As this check also fails for any future ledger, the process will not be able to send validation messages anymore. If this fault manifests in more than f processes, the network does not make progress anymore.

Insufficient Support

A process of the XRP Ledger ensures that it does not diverge from its peers by checking that the ledger it is currently working on has support from a majority of the network. Because the UNLs of our test network only have an overlap of 80%, it is possible for the network to diverge and stall as shown in Figure 5.

Suppose that the first three processes work on ledger A while the last three processes work on ledger B. Both in UNL_1 and UNL_2, the processes will see a majority of at least 60% for the ledger they are working on. Because the fraction of processes in their UNL that works on a different ledger is less than 50% they will not switch to a different ledger, i.e. there is insufficient support and the two UNLs will not converge without manual intervention.

Agreement Violation

A stricter version of the insufficient support bug does not only lead to the processes diverging in the preferred ledger and stalling the network, but results in a fork where the two UNLs fully validate different ledgers as can also be seen in Figure 5.

If in a system where there is already a condition of insufficient support, a Byzantine process that is trusted by both UNLs sends conflicting validation messages, both UNLs can see a quorum of validations for their ledger. They receive two matching validations from their peers in their own UNL, receive one vote from the Byzantine process and count their own vote as well. Since they see four out of five processes agreeing, they fully validate conflicting ledgers.

5.2 Baseline Method

To answer RQ1, we implemented and applied the state-of-the-art statistical bug isolation algorithm [18] as a baseline method to isolate consensus violations in the XRP Ledger and present its output in Table 2. We describe the three identified predicates in order of decreasing importance. First, we explain the context and meaning of the predicate's condition, followed by an evaluation of its expressiveness.

Super-Bug Predictor

The first predicate that was identified is located inside the `hashOfSeq()` function, which, for a given ledger $ledger$ and sequence number seq, either returns $ledger$'s hash or one of its ancestors' hashes, depending on the value of seq. From the predicate's condition and its context, we can infer that the predicate is observed true when the requested seq is at least two less than $ledger$'s sequence number.

Out of the total 145 faulty runs, 142 are attributed to this predicate. The predicate can only discriminate between faulty and successful runs, but not between the underlying bugs. Therefore, we refer to it as a super-bug predictor. While these predictors can be useful to detect the presence of faults, they do not help the developer to debug an execution and identify the underlying root cause.

Non-Predictors

We report the last two predicates as non-predictors because they have no predictive power. Their recall scores do not exceed 2.0% for any bug, meaning that per class of bug, no significant amount of runs is retrieved.

The second-last predicate only has a precision of at most 50.0%, further supporting it being a non-predictor. Neither do the two executions it isolated overlap in their root causes, nor is the condition of the identified if statement related to a bug. The last predicate only isolated a single run and had an importance of zero, making it another non-predictor *a priori*.

We hypothesize that these two predicates are a by-product of the recursive redundancy elimination. After removing all failing runs for which the first predicate was observed to be true, the recursive procedure is left with only three other failing runs. The statistics used to isolate a predicate become inaccurate when a few failing runs are observed, hence, the output towards the end of the algorithm is noisy.

5.3 ISOLATION Algorithm

To answer RQ2, we evaluate our new ISOLATION algorithm against the benchmark data set. Table 3 shows six out of the eight identified predicates. In the following, we will describe and evaluate the different predictors and discuss their performance compared to the baseline method.

Predictor for Insufficient Support Bug

The first predicate returned by ISOLATION is a predictor for the *insufficient support* bug. All of the 101 executions, for which the predicate is observed to be true, share this bug as one of their root causes, giving the predicate a precision of 100.0%. Combined with its high recall of 87.1%, the F_1 and $F_{0.5}$ scores of 93.1% and 97.1% outperform the baseline with 89.1% and 84.1%, respectively.

Further, the predicate's tolerance, threshold, and assertions match Figure 3 and the description in Section 5.1. In this example, the three processes that cause the predicate to exceed its threshold are p_4, p_5, and p_6. Because the two UNLs diverge in their votes, they receive two trusted validations with different ledger hashes for the same sequence number from p_2 and p_3. The assertion that the signing times of the ledger are different maps to the fact that the two UNLs progress independently and are not synchronized anymore. We note that the predicate also holds for other scenarios of the bug, e.g. where p_3 agrees with UNL_2 instead of UNL_1.

Agreement Bug

As stated previously in Section 5.1, the *agreement* bug is a stricter version of the insufficient support bug. It requires an additional round during which a Byzantine process sends conflicting validation messages to the different UNLs.

The resulting failures for the two bugs are different. The insufficient support bug manifests as a violation of liveness, whereas the agreement bug manifests as a violation of safety. However, the two bugs share an underlying root cause: insufficient overlap between the two UNLs.

Since ISOLATION's goal is to discriminate bugs by their root cause, it is correct to group the insufficient support and agreement bug. ISOLATION identified that the two bugs share the same underlying fault and isolated a single predicate for both.

Predictor for Incompatible Bug

The second predicate relates to the *incompatible ledger* bug. It is observed to be true if at least four processes receive at least three validations for a ledger with sequence i signed at time t. This must be followed by a validation with a sequence number j, where $j \neq i$, signed at the same time. Additionally, the predictor correctly captures that the sender of the right-hand validation cannot have previously sent a validation, as this would not cause the bug to expose.

Again, this predictor outperforms the baseline method with F-scores of 58.3% and 77.8% compared to 50.8% and 39.6%, respectively. While the precision is 100.0%, we observe a lower recall score of only 41.2%. On the one hand, we attribute this to 19.6% of executions containing both the insufficient support and incompatible ledger bug, which were already selected by the first predicate. On the other hand, another 25.4% of runs are retrieved by the next two predicates, both sub-bug predictors we described in the following.

The third predicate, being very similar in nature to the second predictor, also targets the incompatible ledger bug. Instead of asserting that the signing time for both validations is the same, it requires the ledgers to be equal. We hypothesize that these conditions are largely equivalent because of a high correlation between the signing time and ledger in a validation. When voting on the contents of a ledger, the processes also vote on its signing time. Therefore, two correct validations for the same ledger share a signing time.

Also, the fourth predicate is similar to the second predictor and only differs in its lower tolerance. This is also observed for the third predicate. We speculate that these differences are a side effect of the recursive redundancy-elimination algorithm. Once a predicate has been removed, the importance of the remaining predicates changes, and previously insignificant predictors can become important.

Non-Predictors

The last four predicates, two of which are shown in Table 3, have little predictive power. Again, this is due to the redundancy-elimination algorithm, which tends to output noise towards the end.

5.4 Summary of Results

Our evaluation has shown that the state-of-the-art baseline method for statistical bug isolation is not capable of discriminating between the consensus bugs in the benchmark data set. The output it generates does not support the developer in identifying the root cause.

We have demonstrated the effectiveness of the ISOLATION algorithm to identify different faults in executions of the XRP Ledger by showing its increased performance compared to the baseline. Our algorithm correctly discriminates between bugs with different root causes. Further, we explained how the predicates relate to the bug triggering conditions and, therefore, sufficiently established ISOLATION's value.

6 Related Work

While, to the best of our knowledge, ISOLATION is the first algorithm to isolate bugs in consensus systems, there is a vast number of publications related to our work [8; 12; 19; 20; 21; 24; 28]. In the following, we will discuss the two most related methods. *Scalable Statistical Bug Isolation* [18] from the field of statistical debugging and *Exposing Complex Bug-Triggering Conditions in Distributed Systems via Graph Mining* [26] from the field of data mining.

The approach proposed in *Scalable Statistical Bug Isolation* depends on a source-to-source compiler that significantly instruments the code to record branching behavior, function's return values, and relations between pairs of scalar variables and values. In comparison, ISOLATION only requires a log of the exchanged messages, which can be easily created using a message interception layer or light instrumentation. The messages sent by the processes sufficiently define the state

Table 3: The results of applying our ISOLATION method to the XRP Ledger. Predicates are ordered by decreasing importance.

Predicate	#Runs	Bug	Precision	Recall	F_1	$F_{0.5}$
$2 \times \texttt{VAL} \xrightarrow[\substack{\mathrm{ledger}(l) \neq \mathrm{ledger}(r) \\ \mathrm{seq}(l) = \mathrm{seq}(r) \\ \mathrm{time}(l) \neq \mathrm{time}(r)}]{\text{for more than 2 processes}} \texttt{VAL}$	101	Incompatible	9.9%	19.6%	13.2%	11.0%
		Insufficient	100.0%	87.1%	93.1%	97.1%
		Agreement	2.0%	100.0%	3.9%	2.5%
$2 \times \texttt{VAL} \xrightarrow[\substack{\mathrm{senders}(l) \cap \mathrm{senders}(r) = \emptyset \\ \mathrm{seq}(l) \neq \mathrm{seq}(r) \\ \mathrm{time}(l) = \mathrm{time}(r)}]{\text{for more than 3 processes}} \texttt{VAL}$	21	Incompatible	100.0%	41.2%	58.3%	77.8%
		Insufficient	28.6%	5.2%	8.8%	15.0%
		Agreement	0.0%	0.0%	0.0%	0.0%
$3 \times \texttt{VAL} \xrightarrow[\substack{\mathrm{senders}(l) \cap \mathrm{senders}(r) = \emptyset \\ \mathrm{ledger}(l) = \mathrm{ledger}(r) \\ \mathrm{seq}(l) \neq \mathrm{seq}(r)}]{\text{for more than 4 processes}} \texttt{VAL}$	4	Incompatible	100.0%	7.8%	14.5%	29.9%
		Insufficient	100.0%	3.4%	6.7%	15.2%
		Agreement	0.0%	0.0%	0.0%	0.0%
$2 \times \texttt{VAL} \xrightarrow[\substack{\mathrm{senders}(l) \cap \mathrm{senders}(r) = \emptyset \\ \mathrm{seq}(l) \neq \mathrm{seq}(r) \\ \mathrm{time}(l) = \mathrm{time}(r)}]{\text{for more than 2 processes}} \texttt{VAL}$	10	Incompatible	90.0%	17.6%	29.5%	49.5%
		Insufficient	10.0%	0.9%	1.6%	3.2%
		Agreement	0.0%	0.0%	0.0%	0.0%
$2 \times \texttt{VAL} \xrightarrow[\substack{\mathrm{senders}(l) = \mathrm{senders}(r) \\ \mathrm{ledger}(l) = \mathrm{prev}(r) \\ \mathrm{time}(l) = \mathrm{time}(r)}]{\text{for more than 1 processes}} \texttt{PRP}$	3	Incompatible	66.7%	3.9%	7.4%	15.9%
		Insufficient	66.7%	1.7%	3.4%	7.8%
		Agreement	0.0%	0.0%	0.0%	0.0%
$2 \times \texttt{VAL} \xrightarrow[\substack{\mathrm{senders}(l) \cap \mathrm{senders}(r) = \emptyset \\ \mathrm{ledger}(l) = \mathrm{ledger}(r) \\ \mathrm{seq}(l) = \mathrm{seq}(r)}]{\text{for more than 1 processes}} \texttt{VAL}$	3	Incompatible	100.0%	5.9%	11.1%	23.8%
		Insufficient	33.3%	0.9%	1.7%	3.9%
		Agreement	0.0%	0.0%	0.0%	0.0%
.... two more predicates omitted						

of the system and, in contrast to the source-based predicates, capture the system's concurrency and nondeterminism.

Although the message patterns, that the graph mining algorithm in *Exposing Complex Bug-Triggering Conditions in Distributed Systems via Graph Mining* generates, specify the interleaving of the messages, they do not assert relations between the bodies of the messages. The algorithm only considers the message as a whole and has no introspect into the individual fields. ISOLATION's predicates however contain binary relationships between the fields, allowing it to isolate more expressive bug predictors for complex protocols.

7 Responsible Research

In conjunction with this paper, we publish our data set and source code on GitHub[1], for other researchers to inspect and reuse. The repository comprises the version of BYZZFUZZ that was used to generate the data set as well as the implementations of the baseline method and our ISOLATION algorithm. While this makes the evaluation of our method fully reproducible, the generation of the data set is, because of BYZZFUZZ, not deterministic. Nevertheless, the generated data tends to show a similar distribution of bugs.

By performing simple validation experiments, we verified the correctness of the statistical framework's implementation, in an effort to limit the risk of bugs in our code. Further, we manually checked the obtained results for consistency.

[1]https://github.com/levinwinter/consensus-bug-isolation

To make our research process transparent, we additionally include the few executions we omitted from the data set. These were removed because they exhibited spurious timeouts that could not be explained by our bug oracles.

The data we collected is purely synthetic and does not contain any real-world transactions. For this, we created a private test-network, running the XRP Ledger, and only processes fabricated transactions. Therefore, the data is free of personally identifiable information.

8 Discussion & Conclusion

Recent testing tools for consensus algorithms return large numbers of buggy executions. To fix these bugs, developers require methods to isolate the underlying faults.

We have empirically shown that our ISOLATION algorithm effectively differentiates different root causes for bugs in consensus systems. The novel predicates we propose successfully capture the bug-triggering conditions and provide explainable predictors for distinct root causes. In addition, we have shown that our method's discrimination abilities outperform the state-of-the-art with higher F_1 and $F_{0.5}$ scores.

ISOLATION supports the analysis of traces that automated testing tools for consensus algorithms generate. The clustering of executions by bug conditions can help developers to identify the underlying root cause more easily.

Future work can investigate how to eliminate or group predicates that are redundant or similar to further improve

ISOLATION's output. In addition, we suggest to devise effective strategies for sampling from the exhaustive set of predicates in an effort to increase ISOLATION's scalability and performance on even larger data sets.

References

[1] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 331–346, New York, NY, USA, 2015. Association for Computing Machinery.

[2] Ignacio Amores-Sesar, Christian Cachin, and Jovana Mićić. Security Analysis of Ripple Consensus. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, volume 184 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[3] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: BFT Systems Made Robust. In Quentin Bramas, Vincent Gramoli, and Alessia Milani, editors, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, volume 217 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:29, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[4] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3):167–178, mar 2010.

[5] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association.

[6] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol.

[7] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 2517–2532, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[8] Hong Cheng, David Lo, Yang Zhou, Xiaoyin Wang, and Xifeng Yan. Identifying bug signatures using discriminative graph mining. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 141–152, New York, NY, USA, 2009. Association for Computing Machinery.

[9] Miguel Correia, Giuliana Santos Veronese, Nuno Ferreira Neves, and Paulo Verissimo. Byzantine consensus in asynchronous message-passing systems: a survey. *International Journal of Critical Computer-Based Systems*, 2(2):141–161, 2011.

[10] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[11] Cezara Drăgoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. Testing consensus implementations using communication closure. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[12] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, page 99–112, New York, NY, USA, 2008. Association for Computing Machinery.

[13] Kyle Kingsbury and Kit Patella. GitHub - jepsen-io/jepsen: A framework for distributed systems verification, with fault injection — github.com. https://github.com/jepsen-io/jepsen.

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

[15] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, May 1998. ACM SIGOPS Hall of Fame Award in 2012.

[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[17] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 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*, OSDI'14, page 399–414, USA, 2014. USENIX Association.

[18] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 15–26, New York, NY, USA, 2005. Association for Computing Machinery.

[19] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News*, 45(1):677–691, 2017.

[20] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 423–437, USA, 2008. USENIX Association.

[21] Alexander V Mirgorodskiy and Barton P Miller. Diagnosing distributed systems with self-propelled instrumentation. In *Middleware 2008: ACM/IFIP/USENIX 9th International Middleware Conference Leuven, Belgium, December 1-5, 2008 Proceedings 9*, pages 82–103. Springer, 2008.

[22] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[23] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.*, 2(OOPSLA), oct 2018.

[24] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault localization in concurrent programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, page 245–254, New York, NY, USA, 2010. Association for Computing Machinery.

[25] David Schwartz, Noah Youngs, Arthur Britto, et al. The Ripple Protocol Consensus Algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

[26] Eunsoo Seo, Mohammad Maifi Hasan Khan, Prasant Mohapatra, Jiawei Han, and Tarek Abdelzaher. Exposing complex bug-triggering conditions in distributed systems via graph mining. In *2011 International Conference on Parallel Processing*, pages 186–195, Sep. 2011.

[27] Levin N. Winter, Florena Buse, Daan de Graaf, Klaus von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.

[28] Maysam Yabandeh, Abhishek Anand, Marco Canini, and Dejan Kostic. Finding almost-invariants in distributed systems. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 177–182, 2011.