Conan: Uncover Consensus Issues in Distributed Databases Using Fuzzing-driven Fault Injection

Haojia Huang
Sun Yat-sen University
huanghj78@mail2.sysu.edu.cn

Pengfei Chen*
Sun Yat-sen University
chenpf7@mail.sysu.edu.cn

Guangba Yu
Sun Yat-sen University
yugb5@mail2.sysu.edu.cn

Haiyu Huang
Sun Yat-sen University
huanghy95@mail2.sysu.edu.cn

Jia Chang
Huawei
Cathy.changjia@huawei.com

Jun Li
Huawei
lijun239@huawei.com

Jian Han
Huawei
hanjian26@huawei.com

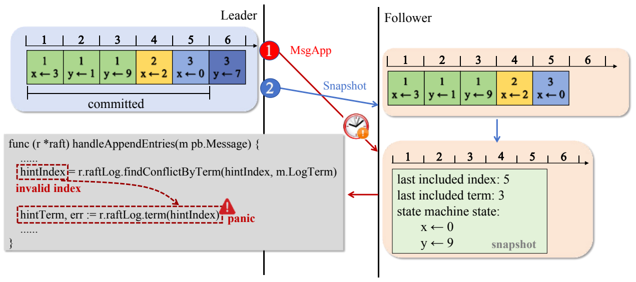
***Abstract*—Consensus is critical for distributed databases as it ensures the consistency of states across nodes, reinforcing the robustness of the overall system. However, faults related to the consensus protocols such as Paxos can lead to serious issues in distributed databases. Such consensus issues impact the correctness and availability of these databases. Therefore, to automatically uncover consensus issues in distributed databases, we propose Conan, a framework designed with fuzzing-driven fault injection. Conan applies a state-guided fuzzing algorithm to effectively explore the fault search space. Moreover, Conan employs hybrid fault sequences that combines fine-grained message-level faults and coarse-grained system-level faults to enhance fault injection. We implement and evaluate Conan on 3 widely-used distributed databases, including etcd, rqlite and openGauss. Finally, Conan has successfully uncovered previously unknown consensus issues, some of which are not detected by existing approaches.**

***Index Terms*—Distributed Databases, Fault Injection, Fuzz Testing, Consensus**

I. Introduction

With the rapid advancement of information technology, distributed architectures have been widely adopted in the database realm [1]–[4]. Distributed databases not only provide efficient computational capabilities but also offer fault tolerance, enhancing system reliability and availability. However, the adoption of distributed databases also introduces new challenges. One critical challenge lies in ensuring consensus among distributed nodes. To address this challenge, most distributed database systems employ consensus protocols such as Paxos [5], [6] or Raft [7]. The crux of these protocols is the efficient exchange of messages between the leader node and follower nodes to reach a consistent view of the system state. However, this process is highly susceptible to disruptions caused by node failures or network anomalies [8]–[10], thereby affecting the consensus process.

To ensure the availability of distributed database, consensus protocols are typically designed with the requirement of preserving data consistency even when partial cluster failures occur. In this study, we define "**consensus issues**" as defects that arise when distributed database nodes fail to maintain consensus under the presence of special cluster failures (e.g., node failures or communication disruptions). As an example,



```
func (r *raft) handleAppendEntries(m pb.Message) {
    ......
    hintIndex = r.raftLog.findConflictByTerm(hintIndex, m.LogTerm)

    hintTerm, err := r.raftLog.term(hintIndex)
    ......
}
```

Fig. 1. A consensus issue in the distributed database *etcd*. This issue arises from network delay fault encountered during the transmission of consensus messages. The root cause of this issue is that the follower's conflict searching algorithm panics unnecessarily when handling a stale MsgApp message (i.e., a message type in the Raft protocol).

Figure 1 shows a real-world consensus issue [11] in the widely-used cloud native database etcd [2]. This issue triggers a process panic, ultimately making etcd unavailable. Additionally, consensus issues in distributed databases can result in more severe consequences, such as degraded user experience, data loss, and even financial losses [12]–[15]. Therefore, uncovering consensus issues is of paramount importance for distributed databases.

However, it is challenging for developers to anticipate and cover all possible fault scenarios in distributed databases, and relying solely on in-house testing is insufficient for exposing consensus issues. Fault injection approaches, which actively introduce faults into database systems, can effectively test the implementations and robustness of consensus protocols. Nonetheless, we identify two primary limitations of existing fault injection approaches in uncovering consensus issues.

- **Neglect of fault injection feedback.** Existing fault injection approaches [16]–[19] employ heuristic-guided or code coverage-guided methods to explore valuable fault injection policies (i.e., fault sequences that indicate when, where and what type of faults to inject). However, they overlook the impact of faults on the system state and neglect the feedback from fault injection. Hence, they struggle to efficiently search through the vast space of fault injection policies.
- **Coarse-grained fault injection.** Existing approaches pri-

marily focus on coarse-grained fault injection at the system level, such as network delay and CPU overload [20]–[22]. However, some consensus issues have stringent trigger conditions, and injecting coarse-grained faults may fail to trigger them. For instance, certain issues may only manifest when a node restarts before receiving a specific message or when messages are delivered out of order.

To overcome the above limitations, we propose **Conan**, a novel fuzzing-driven fault injection approach which can effectively generate fault sequences and reveal consensus issues in distributed databases. Compared with existing fault injection approaches, **Conan** can effectively generate fault sequences based on a state-guided fuzzing algorithm. Specifically, Conan monitors the system's state changes (e.g., role changes, election timeout, etc.) and utilizes them as feedback to guide the selection of seed sequences and mutation operators to generate effective fault sequences. Furthermore, **Conan** considers both coarse-grained system-level faults (e.g., node restart, network packet loss, and network delay) and fine-grained message-level faults (e.g., message loss, message delay, and message term modification) to perform more comprehensive fault injection.

We implement Conan and evaluate its effectiveness on three widely-used distributed databases including etcd [2], rqlite [3] and openGauss [23]. Conan has found multiple previously unknown consensus issues in these systems. We demonstrate the effectiveness of state-guided fuzzing by comparing it with other alternative methods. Additionally, we establish the rationality of the hybrid fault sequences by statistically analyzing the triggering faults of each consensus issue.

To sum up, we make the following contributions.

- We propose a state-guided fuzzing algorithm to generate hybrid fault sequences that combine fine-grained message-level faults and coarse-grained system-level faults to enhance fault injection.
- We propose Conan, a framework for fault injection to uncover consensus issues in distributed databases. We implement and evaluate Conan on three widely-used distributed databases, i.e., etcd, rqlite and openGauss.
- We release Conan [1] as an open-source tool to facilitate the reproduction of discovered issues and to extend it across a broader range of distributed databases.

II. Background and Motivation

A. Background

Distributed Databases. Due to their scalability, fault tolerance and performance, distributed databases are becoming the mainstream type of database [1]–[4]. These databases use Paxos-based consensus protocols [5], [7] to maintain consensus of the nodes and achieve fault tolerance. Consequently, the design and implementation of consensus protocols are crucial for distributed databases. Any defects in the implementation of these protocols can significantly impact the overall performance and availability of the database. Hence, it is essential to find defects in the implementation of consensus protocols. Given the complexity in their implementation, traditional testing methods always prove inadequate [24]–[27], making fault injection a compelling alternative for ensuring robustness.

Fault Injection. Fault injection is a runtime testing technique used to evaluate the resilience and fault tolerance of software systems when they encounter unexpected failures. There are many works aimed at uncovering bugs using fault injection in distributed systems. Notably, Jepsen [28] stands as a widely acknowledged and established tool for conducting fault injection testing. It effectively injects network and node faults, to automatically detect potential issues within the system when subjected to these circumstances. It has inspired numerous works addressing various scenarios based on its technical concepts [29]–[35].

Due to the fact that issues typically are not triggered simply by injecting a single-point fault, existing approaches construct a series of faults into a sequence for injection [16]–[19]. Nevertheless, effectively triggering issues in a sequence of faults necessitates determining the optimal timing and location for their injection. This task is particularly challenging due to the inherent complexity of distributed systems, which results in a vast search space for fault injection.

To tackle this challenge, current approaches utilize fuzzing techniques to identify fault sequences. Initially developed as a conventional software testing method for generating abnormal test inputs [36], the principles of fuzzing can be effectively applied to the generation of fault sequences.

Existing works use heuristic-guided fuzzing (e.g., Chronos [16] and Phoenix [19]) or code coverage-guided fuzzing (e.g., CrashFuzz [17] and FIFUZZ [18]) to explore the faults search space, thereby generating fault sequences. These approaches still encounter certain limitations: (1) Approaches using heuristic-guided fuzzing (e.g., deep-priority rule in Phoenix [19]) lack the flexibility to adapt dynamically and are customized for specific scenarios, exhibiting poor generality. (2) The complexity of system implementation makes accurate code coverage measurement challenging. (3) In some testing scenarios, the increase in code coverage does not necessarily imply the proportional increase in the likelihood of issues [37].

B. Motivation

As discussed in Section II-A, the consensus protocol plays a crucial role in distributed databases. Uncovering consensus issues thoroughly is essential to prevent them from faults, enabling distributed databases to offer more robust services. However, effectively generating fault sequences to uncover consensus issues is a non-trivial task due to inherent limitations. In order to address these limitations, we present two motivations for effectively revealing consensus issues.

Motivation 1: Monitoring state changes helps in generating effective fault sequences. Through our observations of existing consensus issues [11], [38]–[40], we have noted that their occurrence frequently coincides with changes in the system state. For example, as illustrated in Figure 1, one such issue arises when the system takes a snapshot of the

[1] https://github.com/huanghj78/conan

log and subsequently truncates the log index. In addition to this specific example, other state changes can also indicate potential issues, such as node role changes or election timeout due to faults. This observation is well-founded because state changes typically indicate that the system is responding to dynamic conditions or addressing specific challenges. When the system's implementation is flawed or lacks robustness, these state changes can expose underlying defects.

To further validate our observation, we conduct a statistical analysis of the issues on five well-known distributed databases on GitHub. Specifically, we select closed and completed issues and use some keywords related to system state changes (e.g., leader election, log snapshot and node failure) to search for related issues. The specific results are shown in Table I. From Table I, we observe that, on average, 54.9% of issues in distributed databases are related to system state changes. This result suggests a significant association between the occurrence of issues and system state changes in most databases. Moreover, based on our experiment results presented in Table IV, all consensus issues uncovered occurred during system state changes, such as leader election and node failure.

Hence, a fault sequence is effective if it can trigger more system state changes. This motivates us to monitor system state changes and use them as feedback to guide the fuzzing algorithm for the effective generation of fault sequences.

TABLE I
GITHUB ISSUE DISTRIBUTION IN SEVERAL DISTRIBUTED DATABASES.

Database	# Issues		Percentage
	Total	System State Related	
etcd [2]	6408	4289	66.9%
rqlite [3]	488	305	62.5%
ZooKeeper [41]	1967	1094	55.6%
CockroachDB [1]	60293	32272	55.5%
TiDB [4]	15491	5249	33.8%

Motivation 2: Combining fine-grained and coarse-grained fault injection enhances issues detection. Existing fault injection tools [20]–[22] mainly focus on coarse-grained fault injection at system level (e.g., network delay, etc.). Injecting these faults can simulate the system being in a fault scenario so as to uncover its resilience vulnerabilities.

However, relying solely on coarse-grained faults is not sufficient. Consider a real-world issue illustrated in Figure 1. This issue only manifests when a stale MsgApp message arrives at the follower node after the log has been truncated [11]. Specifically, it requires that the MsgApp message sent before the Snapshot message arrives at the follower later than the Snapshot message. Such a scenario occurs during a transient network error. Injecting a coarse-grained network delay affects all messages (including MsgApp and Snapshot), making it challenging to reproduce the specific scenario. In contrast, this scenario can be easily replicated if we can introduce fine-grained delay to only specific messages (i.e., MsgApp). As demonstrated in our experimental findings (Section IV-C2), 71.4% issues cannot be uncovered without fine-grained faults. Therefore, combining fine-grained faults with coarse-grained faults can enhance the ability to uncover consensus issues.

Moreover, fine-grained faults can reduce testing overhead in certain scenarios. For instance, a real-world issue [42] occurs only when the leader restart at term MAX_UINT64. Simulating such a scenario by repeatedly restarting the leader can be time-consuming. Conversely, modifying the term field of the message allows for easier simulation, which can be achieved by taking one over MAX_UINT64 of the time.

Overall, combining fine-grained and coarse-grained faults helps uncover more consensus issues while reducing testing overhead. This motivates us to inject hybrid fault sequences that combine fine-grained message-level faults and coarse-grained system-level faults.

C. Problem Formulation

We formalize the problem of generating fault sequences for consensus issues detection as follows. Consider a distributed database $\mathcal{D}(\mathcal{N}, \mathcal{T})$ consisting of N nodes that communicate to maintain consensus. $\mathcal{T} = \{t_1, t_2, ..., t_n\}$ represents the set of crucial moments in the communication process. We refer to each element in $\mathcal{T}$ as a Fault Trigger Point (*FTP*) (e.g., After Leader Send in Figure 2). The communication between nodes to maintain consensus forms an *FTP* sequence.

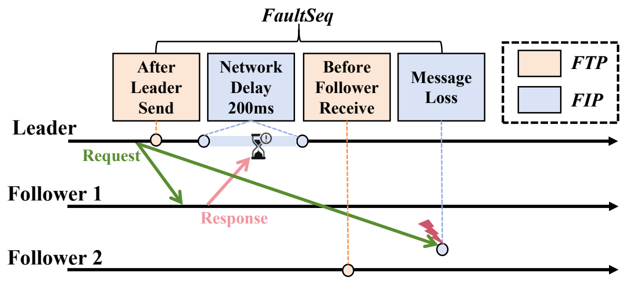


Fig. 2. An example of *FaultSeq*.

Given the set of possible fault injection actions $\mathcal{F}$, where $\mathcal{F} = \{f_1, f_2, ..., f_l\}$ represents the set of faults that may occur at each moment in $\mathcal{T}$. We refer to each element in $\mathcal{F}$ as a Fault Injection Point (*FIP*) (e.g., Message Loss in Figure 2). *FIP*s can be inserted into the *FTP* sequence, resulting in a sequence composed of both *FTP*s and *FIP*s. We define this sequence as a Fault Sequence (*FaultSeq*), which indicates when, where, and what type of faults the database will experience. Figure 2 illustrates how a *FaultSeq* with two *FTP*s and two *FIP*s performs fault injection.

Given the system state $\mathcal{S}$ collected from $\mathcal{D}$, we formalize the fault sequence generation as a parameterized model $\mathcal{M}$:

$$\mathcal{M} : (\mathcal{D}(\mathcal{N}, \mathcal{T}), \mathcal{F}, \mathcal{S}) \rightarrow FaultSeq.$$

In an ideal scenario, a system's correctness and robustness should ensure expected performance even in the presence of a $FaultSeq$. However, in practice, the system may reveal issues when subjected to the effects of $FaultSeq$. If the $FaultSeq$ induces more system state changes than normal, it indicates that it is disrupting the system's state, making it valuable to trigger consensus issues. Hence, Conan's role is to generate valuable $FaultSeq$ and use a workload and oracle to detect issues. An 'oracle' in software testing is a criterion for determining correct system performance [43].

III. Conan Design

A. Design Goal

According to the motivation mentioned in Section II-B, we meticulously design the following features of Conan to effectively uncover consensus issues.

Applying a state-guided fuzzing algorithm (Section III-C). Conan monitors the system state changes and utilizes them as feedback to guide the selection of seed sequences and mutation operators, aiming to generate valuable $FaultSeq$ with a higher probability.

Using hybrid fault sequences (Section III-D). Conan divides fault types into coarse-grained system-level faults and fine-grained message-level faults. System-level faults are faults that affect the entire system for a period of time, such as network delay. Message-level faults are faults that affect a specific message instantaneously, such as message loss.

B. Overview

Conan is a fuzzing-driven fault injection framework designed to uncover consensus issues in distributed databases.

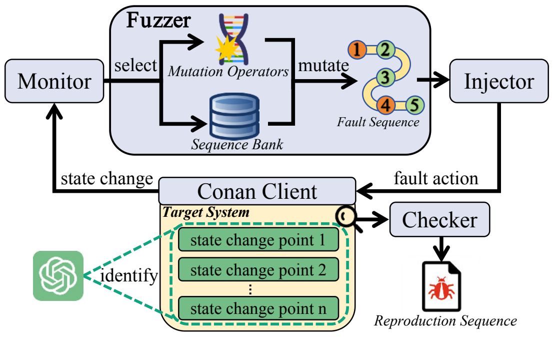


Fig. 3. The overview of Conan.

Components. Figure 3 illustrates the overview of Conan, which comprises five components: Monitor, Fuzzer, Injector, Checker, and Conan Client.

Conan Client is a set of interfaces instrumented within the target system, enabling interaction between the target system and Conan. **State change point** is defined as a specific location within the source code where a system state change occurs. These changes include updates to variables, the invocation of functions that alter object states, and adjustments in system behavior triggered by external events. Given the multitude of potential state change points, we concentrate on two primary categories of state changes: alterations in properties related to the consensus mechanism (e.g., transitions in node roles) and errors associated with the execution of consensus mechanisms (e.g., timeouts during leader election).

State change points in the target system are identified with the assistance of a large language model (LLM). When these state change points are activated, relevant information is relayed to Monitor through Conan Client. **Monitor** is responsible for collecting the state change information and using it as feedback to guide Fuzzer in generating new fault sequences. **Fuzzer** generates fault sequences based on this feedback, while **Injector** is responsible for injecting these faults into the target system through Conan Client. Finally, **Checker** is tasked with detecting issues that arise during the testing process.

Workflow. Algorithm 1 illustrates Conan's overall workflow. In simple terms, by taking the workload of the target system, a testing oracle, and a time budget for testing as input, Conan continuously performs rounds of workload and fault injections within the allocated time. This iterative process generates sequences of fault injections and then uncovers issues based on the specified oracle.

Specifically, Conan starts by initializing the prior probability distribution of the mutation operators which is used for mutating $FaultSeq$ (Line 2) and the seed sequence bank which stores initial $FaultSeq$ (Line 3). The set $SuspiciousSeq$ is initialized as empty to accumulate $FaultSeq$ potentially revealing issues (Line 4). Subsequently, within the allocated time budget, Conan executes the following steps in cycles, referred to as testing rounds. Firstly, Conan selects a $FaultSeq$ and a mutation operator (Line 6-7). Then, it mutates the $FaultSeq$ using the mutation operator, resulting in a new $FaultSeq$, which is utilized for fault injection (Line 8-9). While executing the workload and injecting faults, Conan monitors the system state to calculate the fitness score of the $FaultSeq$ for subsequent selection (Line 10-13). Finally, Conan checks whether the system violates the oracle; if so, the $FaultSeq$ is inserted into $SuspiciousSeq$ (Line 14-15).

Algorithm 1: The workflow of Conan.

```
1  Function RunConan(Workload, Oracle, TimeBudget):
2      $InitMutationOp()$
3      $SeedBank \leftarrow InitializeSeedBank()$
4      $SuspiciousSeq \leftarrow \emptyset$
5      while TimeElapsed < TimeBudget do
6          $SeedSeq \leftarrow SelectSeed(SeedBank)$
7          $MutationOp \leftarrow SelectMutationOp()$
8          $FaultSeq \leftarrow Mutate(SeedSeq, MutationOp)$
9          $State \leftarrow Inject(Workload, FaultSeq)$
10         $FitnessScore \leftarrow Calculate(State)$
11         $UpdateMutationOp(MutationOp, FitnessScore)$

12         if FitnessScore > Threshold then
13             $SeedBank \leftarrow SeedBank \cup FaultSeq$
14         if Check(Oracle) is not pass then
15             $SuspiciousSeq \leftarrow SuspiciousSeq \cup FaultSeq$

16     return SuspiciousSeq
```

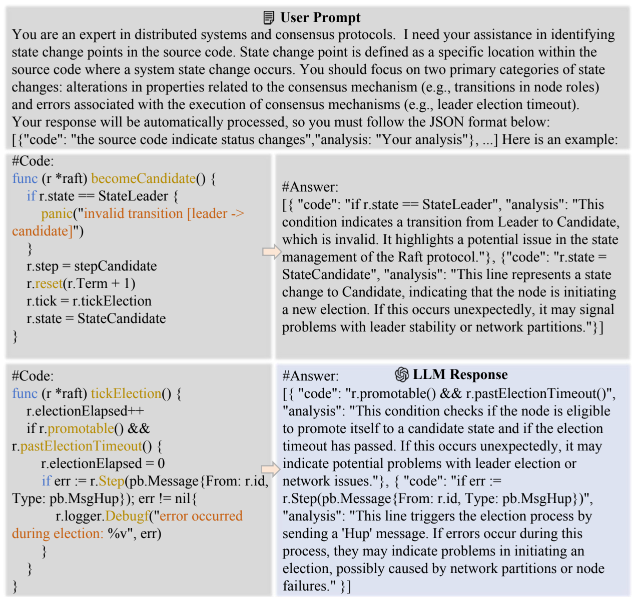
Overall, the entire workflow can be divided into the following three stages: (1) **Fault Sequences Generation** (Section III-C), (2) **Fault Injection** (Section III-D), (3) **Issue Detection and Reproduction** (Section III-E).

C. Fault Sequence Generation

As mentioned in Section II-C, the major task for uncovering more consensus issues is to generate a sufficient number of effective $FaultSeq$. To achieve this goal, Conan employs a state-guided fuzzing algorithm to generate these valuable $FaultSeq$. The initial step involves identifying state change points. During testing, Monitor collects feedback from these points, guiding Fuzzer in selecting seed sequences and mutation operators. The seed sequence can be mutated into a new, valuable $FaultSeq$ through the operations of the mutation operator. Next, we will provide more details on each step.

*1) **State Change Points Identification**:* State change points are utilized to provide feedback to the fuzzing workflow and only need to be identified once, during the deployment of the target system. Due to the complexity of distributed systems, the system state can become numerous. In the case of Raft protocol, the system state includes node roles, log term, committed log entries, the last log index, and so on. Manually identifying the relevant state change points in the source code can be quite challenging. Given the excellent ability of large language models (LLMs) to comprehend code [44], [45], we can leverage them to help us identify state change points in the source code.

To assist in identifying state change points within the source code, we adopt the in-context learning (ICL) paradigm [46], which allows LLMs to generate the expected answers by analogizing the related examples in the prompt demonstration [47], [48]. Specifically, we select code snippets that represent typical state change points according to their functional diversity and provide detailed explanations to serve as examples. Then, we divide the source code into code snippets based on function bodies and present these snippets along with the examples to the LLM for state change point identification. Figure 4 illustrates a simplified prompt template we use in this process, along with a response generated by the LLM.


User Prompt

You are an expert in distributed systems and consensus protocols. I need your assistance in identifying state change points in the source code. State change point is defined as a specific location within the source code where a system state change occurs. You should focus on two primary categories of state changes: alterations in properties related to the consensus mechanism (e.g., transitions in node roles) and errors associated with the execution of consensus mechanisms (e.g., leader election timeout).
Your response will be automatically processed, so you must follow the JSON format below:
[{"code": "the source code indicate status changes","analysis: "Your analysis"}, ...] Here is an example:

```
#Code:
func (r *raft) becomeCandidate() {
   if r.state == StateLeader {
      panic("invalid transition [leader ->
candidate]")
   }
   r.step = stepCandidate
   r.reset(r.Term + 1)
   r.tick = r.tickElection
   r.state = StateCandidate
}
```

#Answer:
[{ "code": "if r.state == StateLeader", "analysis": "This condition indicates a transition from Leader to Candidate, which is invalid. It highlights a potential issue in the state management of the Raft protocol."}, {"code": "r.state = StateCandidate", "analysis": "This line represents a state change to Candidate, indicating that the node is initiating a new election. If this occurs unexpectedly, it may signal problems with leader stability or network partitions."}]

```
#Code:
func (r *raft) tickElection() {
   r.electionElapsed++
   if r.promotable() &&
r.pastElectionTimeout() {
      r.electionElapsed = 0
      if err := r.Step(pb.Message{From: r.id,
Type: pb.MsgHup}); err != nil{
         r.logger.Debugf("error occurred
during election: %v", err)
      }
   }
}
```

LLM Response

#Answer:
[{ "code": "r.promotable() && r.pastElectionTimeout()", "analysis": "This condition checks if the node is eligible to promote itself to a candidate state and if the election timeout has passed. If this occurs unexpectedly, it may indicate potential problems with leader election or network issues."}, { "code": "if err := r.Step(pb.Message{From: r.id, Type: pb.MsgHup})", "analysis": "This line triggers the election process by sending a 'Hup' message. If errors occur during this process, they may indicate problems in initiating an election, possibly caused by network partitions or node failures." }]

Fig. 4. A simplified user prompt and LLM response for state change point identification.

Upon identifying the state change points within the system, we instrument Conan Client at these designated locations to report information to Monitor. During each testing round, Monitor computes a fitness score that reflects the frequency of state changes. This score is derived through the calculation of the dot product of two vectors: the first vector encapsulates the occurrence count of each state change point, while the second vector documents the assigned weights for each state change point, reflecting its relative importance. These weights are set through a combination of domain expert input and iterative optimization based on historical performance data to ensure they accurately reflect the impact of each variable on the system's fitness score. The formula is shown as follows.

$$FitnessScore = \sum_{i=1}^{n} x_i \cdot w_i. \tag{1}$$

*2) **Seed Sequences Selection**:* Fuzzer selects seed sequences from a seed bank and mutates them to generate new fault sequences for fault injection. At startup, Fuzzer randomly generates a set of seed sequences. When a seed sequence is mutated and executed, it obtains a fitness score as described in Section III-C1. If the fitness score exceeds a certain threshold, Fuzzer considers the sequence valuable and adds it to the seed bank. Otherwise, Fuzzer discards it.

When selecting seed sequences, Fuzzer uses fitness scores as selection criteria. We aim to ensure that every sequence in the seed bank has a chance to be selected, while prioritizing those with high fitness scores. Hence, the selection strategy is as follows. Initially, a certain number of candidate sequences are randomly chosen from the seed bank, after which the sequence with the highest fitness score among these candidates is selected. Furthermore, to sustain diversity within the seed bank, each seed sequence will be removed from the seed bank after being used a predetermined number of times.

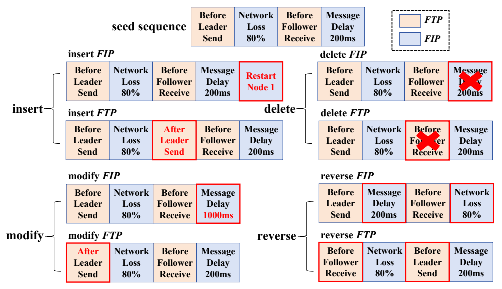

seed sequence
Before Leader Send | Network Loss 80% | Before Follower Receive | Message Delay 200ms
FTP
FIP
insert
insert FIP
Before Leader Send | Network Loss 80% | Before Follower Receive | Message Delay 200ms | Restart Node 1
insert FTP
Before Leader Send | Network Loss 80% | After Leader Send | Before Follower Receive | Message Delay 200ms
delete
delete FIP
Before Leader Send | Network Loss 80% | Before Follower Receive | Message Delay 200ms
delete FTP
Before Leader Send | Network Loss 80% | Before Follower Receive | Message Delay 200ms
modify
modify FIP
Before Leader Send | Network Loss 80% | Before Follower Receive | Message Delay 1000ms
modify FTP
After Leader Send | Network Loss 80% | Before Follower Receive | Message Delay 200ms
reverse
reverse FIP
Before Leader Send | Message Delay 200ms | Before Follower Receive | Network Loss 80%
reverse FTP
Before Follower Receive | Network Loss 80% | Before Leader Send | Message Delay 200ms

Fig. 5. An example of mutation operators.

*3) **Mutation Operators Selection**:* Mutation operators represent a series of operations that mutate seed sequences to generate new fault sequences. Specifically, we use four basic types of mutation operators including **insert**, **delete**, **modify**, **reverse**. The definitions of these operators are as follows.

- **Insert** a new FIP or FTP into the seed sequence.
- **Delete** an FIP or FTP from the seed sequence.
- **Modify** the arguments of an FIP or FTP.
- **Reverse** a pair of FIP or FTP.

Figure 5 shows an example of the four types of operators. Based on these four types of mutation operators, Fuzzer can generate diverse and comprehensive fault sequences.

Different mutation operators hold varying probabilities of generating valuable new sequences for different seed sequences. Therefore, given the variety of mutation operators and seed sequences, selecting the appropriate mutation operator becomes a critical consideration. This can be conceptualized as an exploration-exploitation (EE) problem. Thompson Sampling (TS) algorithm [49] is a simple and effective algorithm to solve EE problem [50]. Algorithm 2 explains how TS is applied for mutation operators selection.

Each operator m has two key attributes: $m.Success$ (success count) and $m.Fail$ (failure count). We use the feedback from Monitor to calculate a $FitnessScore$ to evaluate whether m is successful or not. Specifically, if $FitnessScore$ exceeds a given threshold, we consider that m has performed a successful mutation. Otherwise, it is a failed mutation. These two attributes serve as parameters of the beta distribution associated with m. Upon initialization, both $m.Success$ and $m.Fail$ are set to 1 (Line 3-4). With each update to m, if the $FitnessScore$ of m surpasses a predefined threshold, $m.Success$ is incremented by one, otherwise $m.Fail$ is incremented by one (Line 6-9). When selecting m, we leverage the TS algorithm for decision-making. For each mutation operator m, we sample a value from its beta distribution (Line 11-13). Ultimately, we select the m with the highest sampling value for mutation operation (Line 14).

Algorithm 2: Mutation operators selection algorithm.

```
1  Function InitMutationOp():
2      foreach $m \in MutationOps$ do
3          $m.Success \leftarrow 1$
4          $m.Fail \leftarrow 1$
5  Function UpdateMutationOp(m, FitnessScore):
6      if $FitnessScore > Threshold$ then
7          $m.Success \leftarrow m.Success + 1$
8      else
9          $m.Fail \leftarrow m.Fail + 1$
10 Function SelectMutationOp():
11     foreach $m \in MutationOps$ do
12         $Sample(m) \sim \text{Beta}(m.Success, m.Fail)$
13     $m^* \leftarrow \arg\max_m Sample(m)$
14     return $m^*$
```

D. Fault Injection

Once the $FaultSeq$ generated by Fuzzer is available, Conan will perform fault injection based on it. Next, we will provide a detailed introduction to $FaultSeq$ and how Conan performs fault injection.

*1) **Fault Sequence**:* As mentioned in Section II-C, a fault sequence ($FaultSeq$) consists of several fault trigger points (FTP) and fault injection points (FIP).

Fault Trigger Point (FTP) is a term used to describe the critical moment in the communication process between leader and followers, representing when and where a fault can be injected. We define eight types of FTP as shown in Figure 6. Each FTP comprises three fields. The first point is 'Before Leader Send', the second is 'After Leader Send' and so forth. Taking 'Before Leader Send' as an example, this FTP means a fault will be injected before the leader sends a message.

Fault Injection Point (FIP) describes a type of fault and the corresponding parameters for that fault type. We categorize FIP into two main classes including coarse-grained **system-level** faults and fine-grained **message-level** faults. System-level faults are faults that affect the entire system for a period of time, while message-level faults are faults that affect a specific message instantaneously. Table II shows all the fault types currently supported by Conan.

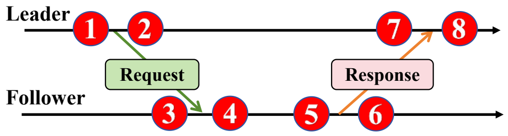


Fig. 6. Fault Trigger Points for leader and follower nodes.

*2) **Fault Injection Mechanism**:* The fault injection mechanism is implemented by Conan Client and Injector.

Conan Client. In order to perform precise and fine-grained fault injection, Conan need to obtain runtime information about the system, such as the current execution flow, types of messages being sent or received, etc. The most straightforward approach to achieve this is code instrumentation. However, code instrumentation inevitably introduces complexity to the system implementation and reduces its generality. Therefore, our goal is to minimize the overhead of code instrumentation.

To accomplish this, a practical method involves offering a suite of interfaces to the target system, treating it as a client, and abstracting the core fault injection and detection logic into an independent server. Specifically, we implement Conan Client as a set of RPC interfaces, interacting with Injector for data flow and control flow. This interaction is achieved by instrumenting these interfaces in the source code related to sending and receiving messages from leader and follower nodes in the target system. Each time the target system reaches an instrumented function point, Conan Client reports runtime information to Injector. Simultaneously, Injector can send tampered messages to Conan Client to influence the original data flow. Since Conan Client employs synchronous RPC, Injector can also manipulate the client's control flow, e.g., simulating message delay by delaying return.

Injector. Injector holds a $FaultSeq$ consisting of FTPs and FIPs to determine when, where, and what type of faults to inject. As $FaultSeq$ is a sequence composed of FTPs and FIPs, Injector can dynamically maintain the current FTP and next FTP based on $FaultSeq$. Each time Injector receives a request from Conan Client, it determines whether the fields in the request match the fields in the current FTP. These fields include the source of the request, the moment of request processing, and the message type. If they match, Injector will execute all FIPs between the current FTP and the next FTP, otherwise wait for the next request to arrive.

E. Issue Detection and Reproduction

After running the workload and injecting faults, Checker detects consensus issues using oracles. We consider a system potentially having issues if it deviates from the oracle. In particular, our focus is on the following four types of deviations.

TABLE II
LIST OF FAULT TYPES SUPPORTED BY CONAN.

FIP Type	Fault Type	Parameters	Description	#
Message-level	message delay	delay time	Insert specific delays in *FTP*.	i
	message loss	-	Drop the specific message.	ii
	message term modified	variation in modifications	Modify message term to simulate recovery after network partition.	iii
System-level	node restart	id	Restart the specified node.	iv
	network loss	loss percent, duration, id	Causes the specified node to experience a network loss.	v
	network delay	delay time, duration, id	Causes the specified node to experience a network delay.	vi

- **Data Inconsistency.** This includes two scenarios: data inconsistency with expectations and data inconsistency among different nodes.
- **Out of Service.** The system becomes unable to provide services.
- **Abnormal Cluster Status.** The system encounters a scenario where multiple leaders exist or no leader is present.
- **Unexpected Output.** The system outputs unexpected results, e.g., exceptions.

Since Injector holds a $FaultSeq$ in each testing round, when Checker identifies a deviation from the oracle after the system executes the workload, it notifies Injector to convert the current $FaultSeq$ into a YAML-formatted file for storage. Subsequently, when starting Conan, the Reproduction mode can be activated by specifying the corresponding YAML file to load. Injector will then use the associated $FaultSeq$ for fault injection, facilitating issue reproduction.

IV. EVALUATION

We implement Conan in Golang, with about 3,000 lines of code (LOC). Additionally, we provide a Conan Client interface library that supports instrumentation for languages such as Golang, C, Java, and Rust, enabling Conan to be compatible with systems implemented in various languages. Furthermore, we write the necessary setup scripts, workload scripts, and oracle scripts for each target system, averaging 400 LOC per system. We employ ChatGLM-4 [51] for the identification of state change points. We implement fine-grained message-level faults through code instrumentation and coarse-grained system-level faults using ChaosBlade [20].

We conduct experiments to investigate the following research questions (RQs).

- **RQ1:** How effective is Conan in uncovering consensus issues in real-world distributed databases?
- **RQ2:** How do the key components of Conan contribute to its effectiveness?
- **RQ3:** What is the overhead of Conan?

A. Evaluation Setup

For our evaluation, we select Jepsen [28] as the baseline tool since it has an established reputation and effectiveness in testing distributed systems. Jepsen is specifically designed to automatically detect issues in distributed databases, making it an ideal choice for our evaluation.

We select etcd [2], rqlite [3] and openGauss [23] as target systems. They are representative distributed databases with varying consensus protocol implementations, detailed in Table III. etcd uses one of the most widely adopted Raft libraries [52] in production environments, which also powers major distributed systems like Kubernetes [53], CockroachDB [1], TiDB [4], and others. rqlite, on the other hand, uses another popular Golang implementation of Raft [54] on Github, which has garnered 8.2k stars. In contrast, openGauss, as an enterprise-level relational database, employs a self-developed distributed consensus framework (DCF) [55] which uses Paxos-based consensus mechanism tailored for high availability and strong consistency.

We use Docker to deploy the target system as a five-node cluster, comprising one leader and four followers. We write a script to execute a specified number of write and read operations as the workload. Each Docker container is equipped with a 12 core CPU, 24 GB memory, and runs on the Ubuntu 18.04 OS. We repeat each experiment for 10 times with the same workload and use the average values for evaluation.

TABLE III
BRIEF DESCRIPTION OF THE TARGET DATABASE SYSTEMS.

System	# Stars	Language	Consensus Protocol
etcd	45.9k	Go	etcd-io/raft [52]
rqlite	14.7k	Go	hashicorp/raft [54]
openGauss	1.3k	C++	openGauss/DCF [55]

B. RQ1: Effectiveness of Conan

*1) **State Change Points Coverage**:* As mentioned in Section II-B, triggering more state changes during testing process implies a higher likelihood of uncovering potential issues. To assess the effectiveness of Conan in comparison to Jepsen, we measured the number of state change points triggered by them across three databases. For each database, we conducted 150 rounds of tests using both Conan and Jepsen. The experiments for each database took an average of 24 hours to complete. Throughout this period, we tally the total number of state change points triggered by each tool.

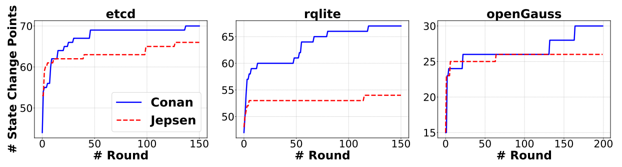


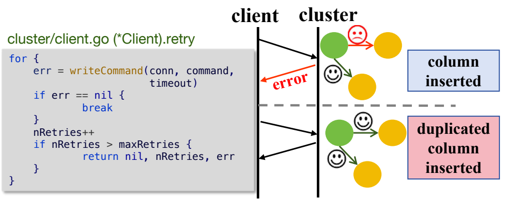
Fig. 7. The trends of state change points coverage for Conan and Jepsen on the three databases.

As shown in Figure 7, the number of state change points triggered by Conan is higher than Jepsen in all three databases, reflecting its superior ability to explore and perturb the system's internal states. This indicates that Conan has greater potential to uncover underlying issues. To further assess their effectiveness, we next compare the actual number of consensus issues uncovered by Conan and Jepsen.

*2) **Consensus Issues Detection**:* As shown in Table IV, Conan has uncovered 7 previously unknown consensus issues, including 1 issue in etcd, 3 issues in rqlite and 3 issues in openGauss. In contrast, Jepsen is only able to uncover three of these issues. We analyze the remaining four issues that can not be uncovered by Jepsen and find that the reason why Jepsen can not uncover them is that their trigger require complex fault sequences or required fine-grained faults. This also further proves the advantage of Conan using state-guided fuzzing algorithm to generate hybrid fault sequences. In the following, we use two cases to illustrate how Conan triggers consensus issues and what consequences these issues will bring.

The first case is the issue #2 in Table IV. The root cause of this issue is the client's unreasonable retry mechanism. When a client sends a write request to a remote node and receives an incorrect response, it initiates a retry. However, this retry mechanism may have a non-idempotent behavior, meaning that it could lead to unintended consequences.

Figure 8 illustrates a scenario that can trigger this issue. When a client sends a write request to a remote cluster, it waits for a response. The cluster can persist data only after reaching a consensus. However, if there is a node experiencing network issues, the process of reaching consensus is prolonged. If the client's timeout time is short, it may trigger a retry due to the timeout, resulting in duplicated data writing.



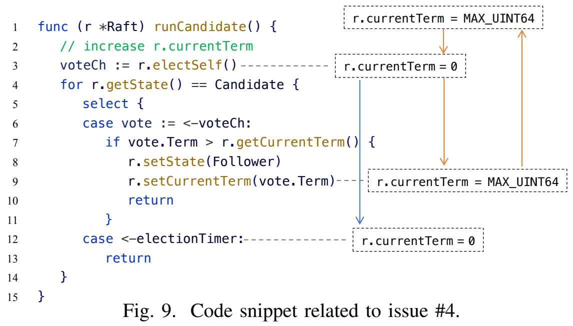
```
for {
    err = writeCommand(conn, command,
                       timeout)
    if err == nil {
            break
    }
    nRetries++
    if nRetries > maxRetries {
            return nil, nRetries, err
    }
}
```

Fig. 8. A buggy scenario related to issue #2.

The second case is issue #4 in Table IV. The root cause of this issue is an overflow of the term field whose type is uint64. For example, in a three-node cluster with nodes labeled node1, node2 and node3, where node1 is the leader, and the current term is MAX_UINT64. If node1 restarts and triggers an election, the following may occur: (1) node2 experiences a heartbeat timeout first, transitions to the candidate state, and executes the code in Figure 9. The invocation of the `electSelf` function (Line 3) increments the `r.currentTerm` variable by 1. Since `r.currentTerm` is MAX_UINT64, it overflows and becomes 0 after incrementing by 1. (2) node2 then awaits a vote from node3. Given node3's `currentTerm` is MAX_UINT64, greater than node2's term, node2 enters the if branch (Line 7). The `setCurrentTerm` function resets node2's term to MAX_UINT64 (Line 9). node2 and node3 will repeat the above process, while node1 will exit as it finds no leader in the cluster.

However, when both node2 and node3 experience a heartbeat timeout simultaneously, transitioning into the candidate state, their term values become 0. In this scenario, neither enters the if branch, waiting until an election timeout to exit (Line 12). Hence, the term will not reach MAX_UINT64 again. In the next election, when a node starts a new election, its term becomes 1, mistakenly making it the leader node.

This case highlights the advantages of Conan in efficiently pushing the system towards the edge case. This aspect is challenging to achieve using traditional fault injection methods.



```
1   func (r *Raft) runCandidate() {
2      // increase r.currentTerm
3      voteCh := r.electSelf()
4      for r.getState() == Candidate {
5         select {
6         case vote := <-voteCh:
7            if vote.Term > r.getCurrentTerm() {
8               r.setState(Follower)
9               r.setCurrentTerm(vote.Term)
10              return
11           }
12        case <-electionTimer:
13           return
14     }
15  }
```

Fig. 9. Code snippet related to issue #4.

Discussion of the two cases. The comparison of these two cases illustrates the rationality of Conan's hybrid fault sequences. Case 1, which simulates coarse-grained faults, emphasizes the system's resilience in the face of overarching faults. In contrast, Case 2 highlights the significance of fine-grained faults, as the faults triggering such issues require finer granularity. Case 2 reveals specific implementation flaws through fine-grained message-level faults. These two cases demonstrate that Conan can assess both the system's overall robustness and its localized weaknesses, offering a more thorough and nuanced evaluation of the system's reliability.

C. RQ2: Evaluation of Key Components

*1) **State-guided Fuzzing**:* Conan uses a state-guided fuzzing algorithm to generate $FaultSeq$. To demonstrate the efficiency of Conan, we compare Conan with other three alternative $FaultSeq$ generation approaches, i.e., BruteForce, Random and Conan-no-feedback. (1) **BruteForce** employs an enumeration strategy to systematically explore all possible combinations of FTP and FIP. Specifically, BruteForce first generates all combinations of FTP and FIP with a ratio of 1:1, then proceeds to generate all combinations with a ratio of 1:2, and so forth. (2) **Random** generates random combinations of FTP and FIP. The length of each random combination is in a specific range. (3) **Conan-no-feedback** is the version of Conan that dose not use state changes as feedback. Specifically, Conan-no-feedback randomly selects seed sequences and mutation operators to generate $FaultSeq$.

We compare the efficiency of these approaches by analyzing the number of testing rounds required to generate $FaultSeq$ capable of triggering each issue. Figure 10 illustrates the number of testing rounds needed by the four approaches to produce the $FaultSeq$ triggering each issue. Fewer rounds means more efficiency. Since Issue #2 and #3 can be trigger from the same $FaultSeq$, we plot them in the same figure.

TABLE IV
THE UNCOVERED CONSENSUS ISSUES.

System	Issue ID	Issue Description	Seq. Len.	Jepsen?	#
etcd	#17332	Inconsistent behaviors between server and client.	2	✗	1
rqlite	#1629	Duplicate data insertion.	1	✓	2
	#1633	Inappropriate error messages.	1	✓	3
	#1712	Unexpected election.	4	✗	4
openGauss	#I8I19W	Data inconsistency between nodes.	2	✓	5
	#I8MGB4	No leader in the cluster.	3	✗	6
	#I8H1YQ	No leader in the cluster.	2	✗	7

In most cases, it is apparent that BruteForce exhibits the lowest efficiency across all scenarios due to the inefficiency of enumeration in large search spaces. The efficiency of Conan is the best among them, and Conan-no-feedback is the second. In comparison to BruteForce, Conan achieves up to a 15x improvement in efficiency at most (Issue #4), with an average improvement of 5.48x.

For Issues #2 and #3, Conan-no-feedback and Conan exhibit slightly lower efficiency compared to Random, possibly attributed to the shorter length of $FaultSeq$ and the simplicity in fault parameter generation. However, Conan consistently outperforms other methods in the remaining cases, with its superiority becoming increasingly evident as the $FaultSeq$ length increases. This further validates Conan's capability to efficiently explore vast search spaces.

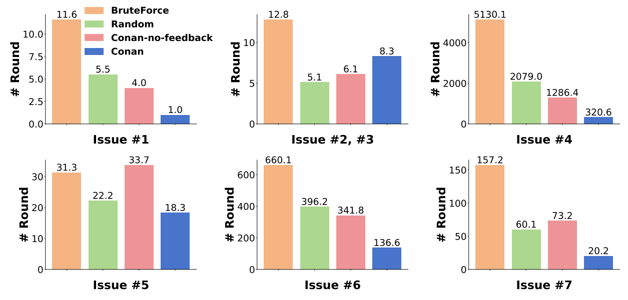


Fig. 10. The testing rounds needed by four approaches to generate *FaultSeq* triggering each issue.

*2) **Hybrid Fault Sequences:*** We collate the faults that contribute to the triggering of each issue and show them in Table V. As we can see, both coarse-grained and fine-grained faults contribute to triggering issues. Some issues can only be triggered by coarse-grained system-level faults (Issue #2, #3), while others can only be triggered by fine-grained message-level faults (Issue #1, #5, #7). Additionally, there are some issues that require the combined effect of system-level faults and message-level faults to trigger (Issue #4, #6). System-level faults and message-level faults complement each other, that is, neither of them can trigger all issues alone, which proves the effectiveness and rationality of the fault types Conan supports.

D. RQ3: Overhead of Code Instrumentation

We instrument the target system with Conan Client to collect information on state changes and perform fault injection. The instrumentation of source code imposes an overhead on both the users and the systems. Hence, it is necessary to evaluate the overhead of the code instrumentation.

TABLE V
THE TRIGGERING FAULTS OF EACH ISSUE.

Issue	Fault Types					
	Message-level			System-level		
	#i	#ii	#iii	#iv	#v	#vi
#1	✓	✓				
#2					✓	✓
#3					✓	✓
#4			✓	✓		
#5	✓					
#6		✓		✓		
#7			✓			

*1) **Overhead for Users:*** Conan provides client interfaces to target system and extracts fault injection logic to Injector to minimize code instrumentation overhead for users. For comparison, we analyze three code instrumentation methods: Conan, Conan-no-interface and Phoenix [19]. Conan-no-interface represents a version of Conan that instruments code directly instead of using Conan Client interfaces. Phoenix is a related approach that injects faults such as network delay by instrumenting fault logic code into the target system.

To ensure fairness, we select fault types supported by both Conan and Phoenix, comparing the increase in compiled code after instrumentation at each injection point. According to our statistics, Conan only requires 5 lines of code (LOC), Conan-no-interface requires 23 LOC, while Phoenix requires 333 LOC. The fewer lines of code that require instrumentation, the lower the cost for users to migrate Conan to a new system, and the less likely they are to encounter issues. Thus, this statistical data suggests that Conan demonstrates strong portability.

*2) **Overhead for Systems:*** We conduct a statistical analysis of three target systems, detailing the original time required for a single write operation and the additional time introduced by Conan instrumentation, as shown in Table VI. It is shown that the instrumentation in the target system incurs an average overhead of approximately 3ms per write operation, which is less than 20% of the original operation time.

Furthermore, leveraging Conan's ability to inject message delay, we assess the impact of delay fault intensity on the systems. For etcd, adverse effects such as request timeouts manifest only when the delay exceeds approximately 1.2s.

Similarly, for rqlite, adverse effects emerge after 30s. In contrast, openGauss follows a design principle prioritizing consistency and partition tolerance over availability in accordance with the CAP (i.e., consistency, availability, and partition tolerance) theorem [56], thus lacking a default timeout mechanism. Hence, we refrain from establishing an impact threshold for openGauss. As shown in Table VI, the impact threshold of the delay faults is much higher than the instrumentation overhead, indicating negligible impact from instrumentation.

In conclusion, Conan can effectively reduce the user's instrumentation overhead. Additionally, the overhead introduced by Conan on system performance can be safely disregarded, as it does not impact the detection of consensus issues.

TABLE VI
INSTRUMENTATION OVERHEAD FOR THE THREE SYSTEMS.

	etcd	rqlite	openGauss
Origin cost	33.2ms	7.79ms	13.6ms
Overhead	4.4ms	1.6ms	3.0ms
Increase	13.3%	20.5%	22.1%
Impact Threshold	1.2s	30s	-

V. DISCUSSION

Limitations. (1) Currently, Conan exclusively injects faults into the message sending and receiving functions associated with the consensus mechanism. Given the complexity of the consensus mechanism, these fault trigger points may not be comprehensive thorough. We will extend Conan to support more fault trigger points, thus conducting a more comprehensive testing of the consensus mechanism. (2) Although Conan endeavors to minimize the cost of code instrumentation, some overhead remains inherent in the instrumentation process. Moreover, if users want to migrate Conan to a new system, they are expected to possess a certain level of domain knowledge about the target system.

Threats to Validity. The internal threat mainly lies in the workloads used by Conan may not diverse enough. To mitigate this, Conan provides a flexible extension method to support other workloads as necessary. The external threat to validity arises from the limited variety of distributed databases chosen for the study. Nonetheless, we strive to be unbiased by selecting systems with different protocol implementations.

VI. RELATED WORK

Software Fault Injection. Software fault injection is a powerful approach to test the resiliency of a system and it can fall into two main categories, namely runtime injection and compile-time injection. **Runtime injection** is primarily implemented by various chaos engineering [57] tools. Netflix proposed Chaos Monkey [21] and Simian Army [22] to enhance the resilience of cloud systems. Alibaba introduced ChaosBlade [20] to inject faults for many application areas. **Compile-time injection** can provide fine-grained fault injection by instrumenting source code. Therefore, many fault injections tailored to specific domains are often implemented through compile-time injection. Sieve [32] instruments the source code of Kubernetes to detect controller bug. WAFFLE [33] instruments the source code of C# applications, introducing delayed injection for MemOrder error detection.

Fuzz Testing. Fuzz testing, or fuzzing, is a software testing technique that involves providing random or unexpected input to a program [36]. Fuzz testing was originally employed to generate test cases, such as GFuzz [58], which detects channel-related concurrency bugs in Go systems by mutating the processing orders of concurrent messages, and Razzer [59], designed to find race bugs in kernels through fuzzing. In recent years, fuzzing has witnessed new developments. For one thing, there has been a growing interest in utilizing large language model (LLM) as fuzzers [60], [61]. For another, researchers have also explored the integration of fuzzing and fault injection to achieve efficient fault injection [17], [18].

Distributed Systems Testing. Given the complexity in the design and implementation of distributed systems, there are currently numerous efforts focused on testing distributed systems. Jepsen [28] is a well-known and general distributed systems testing and verification tool designed to evaluate the reliability and correctness of distributed databases and systems. CoFI [10] injects network partitions to effectively expose partition bugs in distributed system. Chronos [16] injects transient delay to finding timeout bugs in distributed systems. Hermes [35] and Phoenix [19] detects Byzantine Fault Tolerance (BFT) protocol bugs in blockchain system.

Main Differences. Although these related works above provide inspiration and enlightenment for our work, there are significant differences. Conan is a framework designed to inject fault and uncover consensus issues in distributed databases. Unlike traditional fault injection tools, Conan supports both coarse-grained system-level faults and fine-grained message-level faults, enabling a more comprehensive examination of the resilience of distributed systems. Unlike traditional fuzzing, Conan employs a state-guided fuzzing algorithm, which leverages system state changes related to the consensus mechanism as feedback to efficiently generate fault sequences.

VII. CONCLUSION

This paper presents Conan, a fuzzing-driven fault injection framework designed to uncover consensus issues in distributed databases. Conan applies a state-guided fuzzing algorithm to explore the fault search space effectively and uses hybrid fault sequences which combine message-level faults and system-level faults to inject faults more comprehensively. We evaluate Conan on three widely-used distributed databases with different consensus protocols. Conan has successfully uncovered seven previously unknown consensus issues in these databases.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant 62272495, in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2023B1515020054. It is also sponsored by Huawei. The corresponding author is Pengfei Chen.

References

[1] (2024) cockroach github repository. [Online]. Available: https://github.com/cockroachdb/cockroach

[2] (2024) etcd github repository. [Online]. Available: https://github.com/etcd-io/etcd

[3] (2024) rqlite github repository. [Online]. Available: https://github.com/rqlite/rqlite

[4] (2024) Tidb github repository. [Online]. Available: https://github.com/pingcap/tidb

[5] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: https://doi.org/10.1145/279227.279229

[6] ——, "Paxos made simple, fast, and byzantine," in *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, ser. Studia Informatica Universalis, A. Bui and H. Fouchal, Eds., vol. 3. Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9.

[7] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014*, G. Gibson and N. Zeldovich, Eds. USENIX Association, 2014, pp. 305–319. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[8] S. Sondhi, S. Saad, K. Shi, M. Mamun, and I. Traoré, "Chaos engineering for understanding consensus algorithms performance in permissioned blockchains," in *IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress, DASC/PiCom/CBDCom/CyberSciTech 2021, Canada, October 25-28, 2021*. IEEE, 2021, pp. 51–59. [Online]. Available: https://doi.org/10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00023

[9] M. Alfatafta, B. Alkhatib, A. Alquraan, and S. Al-Kiswany, "Toward a generic fault tolerance technique for partial network partitioning," in *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 351–368. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/alfatafta

[10] 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. [Online]. Available: https://doi.org/10.1145/3324884.3416548

[11] (2023) raft: don't panic when looking for term conflicts #36. [Online]. Available: https://github.com/etcd-io/raft/pull/36

[12] (2017) Postmortem of database outage of january 31. [Online]. Available: https://about.gitlab.com/blog/2017/02/10/postmortem-of-database-outage-of-january-31/

[13] (2018) Visa outage: payment chaos after card network crashes – as it happened. [Online]. Available: https://www.theguardian.com/world/live/2018/jun/01/visa-outage-payment-chaos-after-card-network-crashes-live-updates

[14] (2021) Td bank suffered systemwide banking outage, services now recovered. [Online]. Available: https://www.privacy.com.sg/uncategorized/td-bank-suffered-systemwide-banking-outage-services-now-recovered/

[15] (2023) How the dao hack changed ethereum and crypto. [Online]. Available: https://www.coindesk.com/consensus-magazine/2023/05/09/coindesk-turns-10-how-the-dao-hack-changed-ethereum-and-crypto/

[16] Y. Chen, "Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 108–108. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00109

[17] Y. Gao, W. Dou, D. Wang, W. Feng, J. Wei, H. Zhong, and T. Huang, "Coverage guided fault injection for cloud systems," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2211–2223. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00186

[18] Z. Jiang, J. Bai, K. Lu, and S. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 2595–2612. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/jiang

[19] F. Ma, Y. Chen, Y. Zhou, J. Sun, Z. Su, Y. Jiang, J. Sun, and H. Li, "Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 1182–1196. [Online]. Available: https://doi.org/10.1145/3576915.3623071

[20] (2023) Chaosblade github repository. [Online]. Available: https://github.com/chaosblade-io/chaosblade

[21] (2023) Netflix/chaosmonkey github repository. [Online]. Available: https://github.com/Netflix/chaosmonkey

[22] (2023) Netflix/simianarmy github repository. [Online]. Available: https://github.com/Netflix/SimianArmy

[23] (2024) opengauss is an open source relational database management system that is released with the mulan psl v2. with the kernel built on huawei's years of experience in the database field and continuously provides competitive features tailored to enterprise-grade scenarios. [Online]. Available: https://opengauss.org/en/

[24] C. Dragoi, C. Enea, B. K. 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. [Online]. Available: https://doi.org/10.1145/3428278

[25] 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. [Online]. Available: http://www.usenix.org/events/nsdi09/tech/full_papers/yang/yang.pdf

[26] 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. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

[27] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 28:1–28:30, 2018. [Online]. Available: https://doi.org/10.1145/3158116

[28] (2024) Jepsen. [Online]. Available: https://jepsen.io/

[29] (2022) Tipocket github repository. [Online]. Available: https://github.com/pingcap/tipocket

[30] (2019) chaos github repository. [Online]. Available: https://github.com/pingcap/chaos

[31] (2024) openchaos github repository. [Online]. Available: https://github.com/openmessaging/openchaos

[32] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, M. K. Aguilera and H. Weatherspoon, Eds. USENIX Association, 2022, pp. 143–159. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/sun

[33] B. A. Stoica, S. Lu, M. Musuvathi, and S. Nath, "WAFFLE: exposing memory ordering bugs efficiently with active delay injection," in *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*, G. A. D. Luna, L. Querzoni, A. Fedorova, and D. Narayanan, Eds. ACM, 2023, pp. 111–126. [Online]. Available: https://doi.org/10.1145/3552326.3567507

[34] Y. Chen, X. Sun, S. Nath, Z. Yang, and T. Xu, "Push-button reliability testing for cloud-backed applications with rainmaker," in *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*, M. Balakrishnan and M. Ghobadi, Eds. USENIX Association, 2023, pp. 1701–1716. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/chen-yinfang

[35] R. Martins, R. Gandhi, P. Narasimhan, S. M. Pertet, A. Casimiro, D. Kreutz, and P. Veríssimo, "Experiences with fault-injection in a byzantine fault-tolerant protocol," in *Middleware 2013 - ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*, ser. Lecture Notes in Computer Science, D. M. Eyers and K. Schwan, Eds., vol.

8275. Springer, 2013, pp. 41–61. [Online]. Available: https://doi.org/10.1007/978-3-642-45065-5_3

[36] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[37] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox fuzzing of distributed systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, W. Meng, C. D. Jensen, C. Cremers, and E. Kirda, Eds. ACM, 2023, pp. 1615–1629. [Online]. Available: https://doi.org/10.1145/3576915.3623097

[38] (2023) Panic occurs when etcd (ver 3.5.13) new node joins the cluster. [Online]. Available: https://github.com/etcd-io/etcd/issues/17855

[39] (2022) Member remains learner after promotion. [Online]. Available: https://github.com/etcd-io/etcd/issues/18347

[40] (2023) leader not found, after slave gets killed. [Online]. Available: https://github.com/rqlite/rqlite/issues/1819

[41] (2024) zookeeper github repository. [Online]. Available: https://github.com/apache/zookeeper

[42] (2023) Leader election times seem too long with max current term #1712. [Online]. Available: https://github.com/rqlite/rqlite/pull/1712

[43] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012. [Online]. Available: https://doi.org/10.1007/978-3-642-29044-2

[44] H. Li, Y. Hao, Y. Zhai, and Z. Qian, "Enhancing static analysis for practical bug detection: An llm-integrated approach," *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA1, pp. 474–499, 2024. [Online]. Available: https://doi.org/10.1145/3649828

[45] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. [Online]. Available: https://openreview.net/forum?id=iaYcJKpY2B_

[46] Q. Dong, L. Li, D. Dai, C. Zheng, Z. Wu, B. Chang, X. Sun, J. Xu, L. Li, and Z. Sui, "A survey for in-context learning," *CoRR*, vol. abs/2301.00234, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2301.00234

[47] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *CHI '21: CHI Conference on Human Factors in Computing Systems, Virtual Event / Yokohama Japan, May 8-13, 2021, Extended Abstracts*, Y. Kitamura, A. Quigley, K. Isbister, and T. Igarashi, Eds. ACM, 2021, pp. 314:1–314:7. [Online]. Available: https://doi.org/10.1145/3411763.3451760

[48] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html

[49] S. Agrawal and N. Goyal, "Analysis of thompson sampling for the multi-armed bandit problem," in *COLT 2012 - The 25th Annual Conference on Learning Theory, June 25-27, 2012, Edinburgh, Scotland*, ser. JMLR Proceedings, S. Mannor, N. Srebro, and R. C. Williamson, Eds., vol. 23. JMLR.org, 2012, pp. 39.1–39.26. [Online]. Available: http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf

[50] M. Crepinsek, S. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 35:1–35:33, 2013. [Online]. Available: https://doi.org/10.1145/2480741.2480752

[51] (2024) Chatglm. [Online]. Available: https://chatglm.cn/

[52] (2024) The raft library of etcd. [Online]. Available: https://github.com/etcd-io/raft

[53] (2023) Kubernetes. [Online]. Available: https://github.com/kubernetes/kubernetes

[54] (2023) The raft library of rqlite. [Online]. Available: https://github.com/hashicorp/raft

[55] (2024) Distributed consensus framework. [Online]. Available: https://gitee.com/opengauss/DCF

[56] E. A. Brewer, "Towards robust distributed systems," in *PODC*, vol. 7, no. 10.1145. Portland, OR, 2000, pp. 343–477.

[57] A. Basiri, N. Behnam, R. de Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35–41, 2016. [Online]. Available: https://doi.org/10.1109/MS.2016.60

[58] Z. Liu, S. Xia, Y. Liang, L. Song, and H. Hu, "Who goes first? detecting go concurrency bugs via message reordering," in *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, B. Falsafi, M. Ferdman, S. Lu, and T. F. Wenisch, Eds. ACM, 2022, pp. 888–902. [Online]. Available: https://doi.org/10.1145/3503222.3507753

[59] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 754–768. [Online]. Available: https://doi.org/10.1109/SP.2019.00017

[60] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang, "Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 423–435. [Online]. Available: https://doi.org/10.1145/3597926.3598067

[61] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case generators: Crafting unusual programs for fuzzing deep learning libraries," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623343