# Greybox Fuzzing of Distributed Systems

Ruijie Meng[*]
National University of Singapore
Singapore
ruijie_meng@u.nus.edu

George Pîrlea[*]
National University of Singapore
Singapore
gpirlea@comp.nus.edu.sg

Abhik Roychoudhury[†]
National University of Singapore
Singapore
abhik@comp.nus.edu.sg

Ilya Sergey
National University of Singapore
Singapore
ilya@nus.edu.sg

## ABSTRACT

Grey-box fuzzing is the lightweight approach of choice for finding bugs in sequential programs. It provides a balance between efficiency and effectiveness by conducting a biased random search over the domain of program inputs using a feedback function from observed test executions. For distributed system testing, however, the state-of-practice is represented today by only black-box tools that do not attempt to infer and exploit any knowledge of the system's past behaviours to guide the search for bugs.

In this work, we present Mallory: the first framework for grey-box fuzz-testing of distributed systems. Unlike popular black-box distributed system fuzzers, such as Jepsen, that search for bugs by randomly injecting network partitions and node faults or by following human-defined schedules, Mallory is *adaptive*. It exercises a novel metric to learn how to maximize the number of observed system behaviors by choosing different sequences of faults, thus increasing the likelihood of finding new bugs. Our approach relies on *timeline-driven testing*. Mallory dynamically constructs Lamport timelines of the system behaviour and further abstracts these timelines into *happens-before summaries*, which serve as a feedback function guiding the fuzz campaign. Subsequently, Mallory reactively learns a policy using Q-learning, enabling it to introduce faults guided by its real-time observation of the summaries.

We have evaluated Mallory on a diverse set of widely-used industrial distributed systems. Compared to the start-of-the-art black-box fuzzer Jepsen, Mallory explores 54.27% more distinct states within 24 hours while achieving a speed-up of 2.24×. At the same time, Mallory finds bugs 1.87× faster, thereby finding more bugs within the given time budget. Mallory discovered 22 zero-day bugs (of which 18 were confirmed by developers), including 10 new vulnerabilities, in rigorously-tested distributed systems such as Braft, Dqlite and Redis. 6 new CVEs have been assigned.

## 1 INTRODUCTION

Fuzz testing or *fuzzing* is a popular technique for finding security vulnerabilities in software systems [9]. At a high level, it involves feeding generated inputs into an application with the goal of finding crashes. Fuzzing can involve a blackbox approach, where inputs are generated in a purely random fashion, or it can be guided by knowledge of the program's internal structure (white-box). The most popular fuzzers are *grey-box*, where the search is guided by run-time observations of program behaviour, collected, as tests execute, for artefacts instrumented at compile time. Thanks to the ease of its deployment and use, grey-box fuzzing is the state-of-the-practice for automatically discovering bugs in sequential programs.

A common approach to finding bugs in distributed systems in practice is *stress-testing*, in which the system is subjected to faults (*e.g.*, network partitions, node crashes) and its behaviour is checked against a property-based specification. This approach is implemented by tools like Jepsen [23], a testing framework that is well-known for its effectiveness in finding consistency violations in distributed databases [29]. An alternative to stress-testing is *systematic testing*, commonly known as *software model checking*. In this approach, the system under test is placed in a deterministic event simulator and its possible schedules are systematically explored [13, 27, 28, 49]. The simulator exercises different interleavings of system events by reordering messages and injecting node and network failures. Systematic testing is well suited to finding "deep" bugs, which require complex event interleavings to manifest, but is relatively heavyweight, as it requires integration with the system under test either in the form of a manually-written pervasive test harness or a system-level interposition layer. While not as effective at finding deep bugs, stress-testing is widely used due to its low cost of adoption and good effort-payout ratio.

***Problem statement***. We make the following observation: in terms of the ease-of-use/effectiveness trade-off, black-box fuzzing of sequential programs is similar to stress-testing of distributed systems, while white-box fuzzing corresponds to software model checking. However, unlike in the sequential case, there is no grey-box fuzzing approach for distributed systems. Our goal is to explore this opportunity by extending Jepsen with the ability to *perform observations* at runtime about the behaviour of the system and to *adapt* its testing strategy based on feedback derived from those observations. In doing so, we are not aiming to match the thoroughness of systematic testing, but to provide a more effective and principled way to conduct stress-testing while maintaining its ease of use.

***Challenges***. In the last decade, developing a grey-box fuzzer for sequential programs has become more streamlined, due to fuzzers like AFL [48]. In short, AFL works by generating and mutating inputs to a program being tested, aiming to trigger crashes or other unexpected behavior. It uses a feedback-driven approach, keeping track of inputs that cause the program to take *new code paths* and

[*] Joint first authors
[†] Corresponding author

*CCS'23, Nov 26–30, 2023, Copenhagen, Denmark*
2023. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

prioritizing mutations that are likely to explore these paths further. Attempting to adapt the greybox approach to Jepsen-style distributed system testing leads us to three questions:

**Q1** What is the space of inputs to a distributed system that could be explored adaptively?
**Q2** What observations are relevant for a distributed system and how should they be represented?
**Q3** How can one obtain feedback from the observations?

Question **Q1** is already answered by Jepsen: the role of "inputs" for distributed systems is played by *schedules*, that can be manipulated by injecting faults. Even though Jepsen can control the fault injection, in the absence of a good feedback function, it (a) requires human-written generators to explore the domain of schedules if something more than random fault injection is required [2] and (b) repeatedly explores equivalent schedules.

To answer **Q2** we recall perhaps the most popular graphical formalism to represent interactions between nodes in distributed systems: so-called *Lamport diagrams* (aka *timelines*), *i.e.*, graphs showing relative positions of system events as well as causality relations between them [15, 24, 31]. Such diagrams have been used in the past for visualizing executions in distributed systems [8]. Our discovery is that they also can be used as distributed analogues of "new code paths" from sequential grey-box fuzzing. In other words, being able to observe and record new shapes of Lamport diagrams is an insight that brings AFL-style fuzzing to a distributed world.

To make our approach practical, we also need to address **Q3**. The problem with using observed Lamport diagrams to construct a feedback function is that in practice no two different runs of a distributed system will produce the *same* timeline. That is, such new observations will *always* produce new feedback, even though in practice many runs are going to be equivalent for the sake of testing purposes—something we need to take into account.[1] As a solution, we present a methodology for extracting feedback from dynamically observed timelines by *abstracting* them into concise *happens-before summaries*, which provide the desired trade-off between the feedback function's precision and effectiveness.

***Contributions***. The solutions to **Q1**–**Q3** provide a versatile conceptual framework for grey-box fuzzing of distributed systems. Building on these insights, we present our main practical innovation: Mallory, the first grey-box fuzzer for distributed systems. Unlike the black-box testing approach of Jepsen that requires human-written schedule generators, Mallory reactively *learns* them by (a) observing the behaviour of the system under test as it executes and (b) rewarding actions that uncover new behaviour. Below, we detail the design, implementation, and evaluation of Mallory.

- *Timeline-driven testing*, a novel fuzzing approach suited for distributed systems: It is based on dynamically constructing Lamport diagrams (timelines) of the system under test as it runs, and further abstracts the timelines into happens-before summaries. It is used to define a *feedback function* guiding grey-box fuzzing.
- *Reactive fuzzing*, a reactive method for making optimal decisions to achieve maximized behaviour diversity: It reactively learns a policy using Q-learning to decide what actions to take in observed states, to incrementally construct a schedule.

```
int membershipRollback(struct raft *r){
  ...
  // Fetch the last committed configuration entry
  entry = logGet(&r->log, r->config_index);
  assert(entry != NULL);
}

static int deleteConflictingEntries(){
  ...
  // Possibly discard uncommitted config changes
  if (uncommitted_config_index >= entry_index){
    rv = membershipRollback(r);
  }
}
```

**Figure 1: Simplified Dqlite code for membership rollback.**

- *End-to-end implementation* of Mallory, a fuzzing framework for distributed systems: Mallory extends the widely used Jepsen framework—Mallory can be seen as an adaptive generator of schedules for Jepsen tests. The tool is publicly available at https://github.com/dsfuzz/mallory.
- *Comprehensive evaluation* of Mallory on several widely-used industrial distributed system implementations. In our experiments, Mallory covers 54.27% more distinct states within 24 hours and achieves the same state coverage about 2.24× faster than Jepsen. In terms of reproducing existing bugs, Mallory speeds up the bug finding by 1.87× and finds 5 more bugs compared to Jepsen. Moreover, in rigorously-tested distributed systems, Mallory found 22 previously unknown bugs, including 10 new security vulnerabilities and 6 newly assigned CVEs. Out of these 22 bugs, 18 bugs have been confirmed by their respective developers. In our experiments, Jepsen could only detect 4 of these bugs.

## 2 OVERVIEW

In this section, we illustrate the workflow of our technique for adaptively detecting anomalies in distributed systems.

### 2.1 Bugs in Distributed Systems

As a motivating example, let us consider a known bug in the implementation of the Raft consensus protocol [34] used by Dqlite, a widely-used distributed version of SQLite developed by Canonical.[2]

The purpose of using a *consensus protocol* in a distributed system is to ensure the system maintains a consistent and reliable state even in the presence of faults. In Raft, one of the most widely used consensus protocols, a single leader accepts client requests and replicates them to all nodes that persist them as log *entries*. Conflicting entries in a Raft cluster can appear when different nodes receive different log entries during a network partition. Over time, the number of replicated entries might grow very large, which, in turn, might cause issues if certain nodes need to be brought up-to-date after having experienced a temporary downtime. To address this issue, Raft periodically takes a *snapshot* of the current system state, discarding old log entries whose outcome is reflected in the snapshot. Additionally, nodes can be removed from the cluster or join it, thus changing the configuration of the system as it runs. When a configuration change is initiated, the current leader replicates a *configuration change entry* to all the nodes in the cluster.

[1] A sequential analogue of concrete distributed timelines would be a trace of *all* memory operations—too precise to recognise equivalent executions.

[2] Available at https://dqlite.io; 3.4k stars on GitHub at the time of writing.

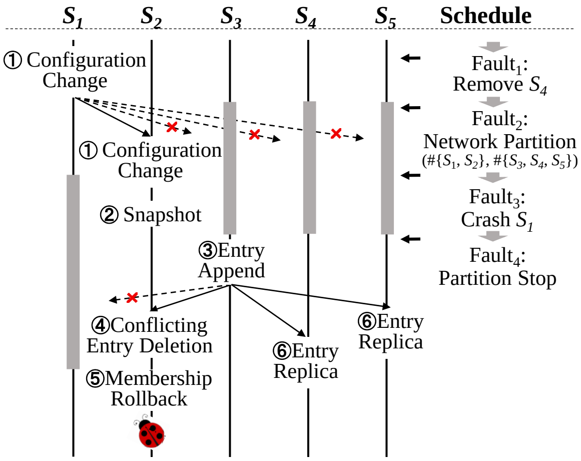

**Figure 2: A timeline of the Dqlite membership rollback bug. Gray vertical rectangles correspond to node downtimes.**

A new configuration becomes permanent once it has been agreed upon and committed by a majority of the nodes, yet a server starts using it as soon as the configuration entry is added to its log, even before it is committed [34, §6]—a fact that is important for our example. If there is a failure during the process of agreeing on a new configuration, such as a network partition, the new configuration may not be fully replicated to the majority, in which case the leader node will attempt to perform a *membership rollback* by adopting the last committed configuration entry from its log.

The bug in question occurs during a membership rollback happening *immediately after* performing a snapshot operation, leading to a failure to restore the last committed configuration [19]. Fig. 1 shows the affected fragment of the actual implementation in Dqlite, which deals with removing conflicting entries during node recovery. In case there is an uncommitted configuration entry among the conflicting entries to remove, a Dqlite server has to first roll back to the previously committed membership configuration via `membershipRollback` (line 1009). When this happens after a snapshot operation, which has removed the last committed configuration entry, the assertion on line 160 gets violated.

To show how this rather subtle bug can be triggered in a real-world environment, consider a run of a Dqlite cluster depicted in Fig. 2. The initial cluster comprises five servers $S_1$–$S_5$, with $S_1$ assumed to be a leader. Server $S_4$ requests (to the leader $S_1$) to be removed from the cluster. Upon receiving this request, leader $S_1$ appends the configuration change entry into its log (①) and attempts to replicate it to all other members, but only succeeds to do so for $S_2$ (①), failing to reach $S_3$, $S_4$, and $S_5$ due to a sudden partition in the network. At the same time, the number of log entries at the server $S_2$ reaches a threshold value, prompting it to take a snapshot (②), while *already using* the latest configuration (which has not been agreed upon by the majority), thus, discarding the old configuration entry from the snapshot. To make things worse, $S_1$ crashes at the same time. Although the network of $S_3$, $S_4$, and $S_5$ recovers soon after, the cluster has already lost its leader. At some point, a new leader election is initiated (details omitted), with $S_3$ eventually becoming the leader and attempting to synchronise logs across the nodes (③). Prompted to do so, $S_2$ detects a conflicting entry in its log (*i.e.*, the uncommitted configuration change (①)) and deletes it (④). It then attempts to retrieve the last committed configuration entry to roll back the membership (⑤), which is long gone due to the prior snapshotting (②), triggering the assertion violation at line 160 of Fig. 1.

**Algorithm 1:** Fuzzing with Jepsen and Mallory

```
Input: $P_0$: system under test (SUT)
Input: $Nem$, $Faults$: a nemesis and the faults it can enact
Input: $Oracles$: a set of test oracles for bug detection
Input: $S$: number of steps in each schedule
Input: $T$: total time budget for testing
Output: $Bugs$: a set of bugs detected
1  $P_f \leftarrow$ instrumentSystem ($P_0$)
2  $Policy \leftarrow \{$ $initState$ , $Faults\}$
3  repeat
4      $curState \leftarrow initState$
5      repeat
6          $fault \leftarrow Policy$.getNextFault ( $curState$ )
7          $Nem$.enactFault($fault$)
8          $events \leftarrow$ observeSystemUnderTest ($P_f$)
9          $timeline \leftarrow$ constructTimeline ($events$)
10         $nextState \leftarrow$ abstractTimeline ($timeline$)
11         $rwd \leftarrow$ calculateReward ($curState$, $fault$, $nextState$)
12         $Policy \leftarrow$ learn ($Policy$, $curState$, $fault$, $rwd$)
13         $curState \leftarrow nextState$
14     until maximum steps $S$ reached
15     resetSystemUnderTest ($P_f$)
16 until time budget $T$ exhausts
17 $Bugs \leftarrow Oracles$.identifyBugs ($events$)
```

## 2.2 Fuzzing Distributed Systems via Jepsen

As demonstrated by the example, identifying a bug in a distributed system in some cases boils down to constructing the right sequence of *faults*, such as network partitions and node removals, resulting in an execution that leads to an inconsistent state and, subsequently, to the violation of a code-level assertion or of an externally observable notion of consistency (*e.g.*, linearisability [18]). The state-of-the-art fuzzing tool Jepsen provides a means to randomly generate sequences of faults with the goal of discovering such bugs.

Algorithm 1 provides a high-level overview of the workings of Jepsen (let us ignore the grayed fragments for now). Jepsen requires a lightweight harness for the system under test to define how to start and stop it, to enact faults, introduce client requests, and collect logs (line 1). For brevity, Algorithm 1 does not show the set-up of the SUT at the beginning of each test or the introduction of client requests, which the SUT is constantly subjected to by client processes. Importantly, Jepsen allows the user to define testing *policies* (aka "generators") responsible for introducing specific types of external inputs or faults (line 2). The main fuzzing loop of Jepsen is shown in lines 3–16 of the algorithm.

During each run of the outer loop, the framework generates a system-specific external input or fault (line 6) via a *policy*, and enacts it using a *nemesis*—a special process, not bound to any particular node, capable of introducing faults. Such inputs may, for example, be the decision to remove a node from the system, as, *e.g.*, is done by node $S_4$ in our running example. As the system is executing, the framework records its observations (line 8) for future analysis to detect the presence of bugs or specification violations (line 17). This process continues until the time budget $T$ is exhausted (line 16). The test run is segmented into schedules of $S$ steps each, after which the system is reset (line 15).

Getting back to our example, we can see that the membership rollback bug can be exposed by the scheduled sequence of inputs/-faults that first initiates the removal of $S_4$ from the cluster and then creates a network partition ($\#\{S_1, S_2\}$, $\#\{S_3, S_4, S_5\}$), followed by node a crash of $S_1$. Randomly generating this particular sequence of faults via Jepsen, while possible, is somewhat unlikely. The reason is: before coming across this schedule, Jepsen may try many others, each making very little difference to the system's observable behaviour, *e.g.*, by randomly crashing a number of nodes. In our experiments, Jepsen failed to detect this membership rollback bug (*i.e.*, Dqlite-323 in Sec. 4.3) within 24 hours.

However, with just a little insight into the system, one can conjecture that enacting a partition right after a configuration change leads to novel system states more often than, *e.g.*, performing another configuration change, thus, increasing the likelihood of witnessing a new, potentially bug-exposing, behaviour. Our goal is to retrofit Jepsen so it could derive these insights at run time and adapt the policies accordingly.

### 2.3 Learning Fault Schedules from Observations

The high-level idea behind Mallory, our fuzzing framework, is to enhance Jepsen with the ability to *learn* what kinds of faults and fault sequences are most likely going to result in previously unseen system behaviours. To achieve that, we augment the baseline logic of Algorithm 1 by incorporating the grayed components that keep track of the observations made during the system runs. The first change is to add instrumentation to the system under test (line 1) to record significant *events* (*e.g.*, taking snapshots or performing membership rollbacks in Fig. 2) during the execution, additional to those Jepsen already records, *i.e.*, client requests, and responses. More interestingly, the fault injection policy is now determined not just by the kinds of faults and inputs that can be enacted, but by the latest *abstract state* of the system, whose nature will be explained in a bit and that is taken to be some default *initState* at the start of the fuzzing campaign (line 2).

The main addition consists of lines 9-13 of the algorithm. Now, while running the system, the fuzzer collects sequences of events recorded by the instrumented nodes, as well as message-passing interactions between them; the exact nature of events and how they are collected will be described in Sec. 3.1. The information about the recorded events and their relative ordering is then used to construct a (Lamport-style) *timeline* and subsequently *summarised* to obtain the new abstract state *nextState* (lines 9-10)—the design of these two procedures, detailed in Sec. 3.2, is the central technical contribution of our work. The newly summarised abstract state is used to calculate the reward *rwd* by estimating how dissimilar it is compared to abstract states observed in the past (line 11). Finally, the reward is used to dynamically update the policy, after which the loop iteration repeats with the updated abstract state (lines 12-13).

Postponing until Sec. 3 the technicalities of computing abstract states, calculating rewards, and updating the policy, let us discuss how the introduced changes might increase the likelihood of discovering the bug-inducing system behaviour from Fig. 2. We now pay attention to the six kinds of events (①-⑥) that can be recorded in the system, as well as their relative happens-before ordering is computed across multiple nodes. Consider a fault injection policy that introduces a sequence of node removals (such as $\text{Fault}_1$). After triggering several configuration changes (*i.e.*, event ①), such a policy will not introduce many new behaviours in a long run, which will prompt our adaptive fuzzer to prefer other faults, *e.g.*, network partitions. By iterating this process, observing new behaviours (*i.e.*, different event sequences) in the form of novel abstract states and de-prioritising policies that have not generated new behaviours, the fuzzer will eventually discover a sequence of faults leading to the membership rollback bug.

It is important to note that the fact that a particular policy has not produced a new abstract state (*i.e.*, a new observable behaviour) in a particular run does not necessarily mean that it needs to be discarded for good. Due to the nature of the applications under test, Mallory, similarly to Jepsen, does not provide a fully deterministic way to inject faults, hence some behaviours might depend on the absolute timing of faults. This is taken into account by Mallory's learning (*cf.* Sec. 3.3), which leaves a possibility for such a policy to be picked again in the future, albeit, with a lower probability.

In our experiments, due to Mallory's adaptive learning, the membership rollback bug was discovered in 8.68 hours (Jepsen failed to discover it in 24 hours). In the following, we give a detailed description of Mallory's design (Sec. 3) and provide thorough empirical evidence of its effectiveness and efficiency for discovering non-trivial bugs in distributed systems (Sec. 4).

## 3 THE MALLORY FRAMEWORK

At its core, Mallory implements an adaptive *observe-orient-decide-act* (OODA) loop:

- *Observe*—observe each node's internal behaviour and intercept all network communication between nodes;
- *Orient*—construct a global Lamport timeline of the system's behaviour to obtain a bird's eye view of the execution, and abstract the timeline into a manageable representation, called a happens-before summary, used to understand the current state of the system and to determine the effectiveness of previous actions;
- *Decide*—choose a fault to inject based on the current observed summary and the past execution history;
- *Act*—inject the fault and repeat the loop.

Unlike white-box fuzzers, which rely on encapsulating the system under test in an event simulator [13, 36], Mallory operates on the actual system in its normal distributed environment—a firm requirement to minimise the friction (*i.e.*, adoption effort). In particular, Mallory does not have the luxury of being able to "pause" the system and observe its state before deciding what actions to take, as it operates in real-time, in a reactive manner. This means Mallory itself is a distributed system, which complicates its implementation

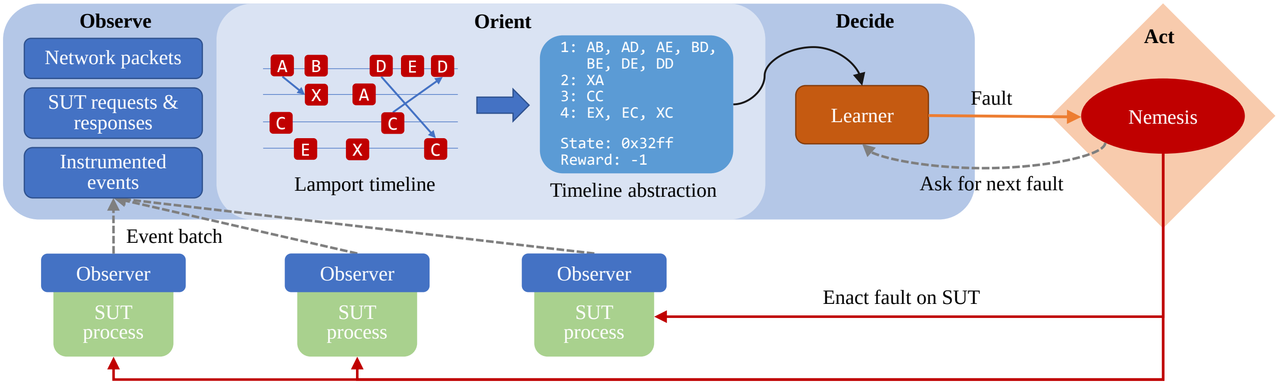

**Figure 3: The central observe-orient-decide-act loop in Mallory. A centralised *mediator* collects events from *observers* distributed at the nodes in the SUT, and drives the test execution. Faults decided by Mallory are enacted by Jepsen.**

slightly. Nonetheless, its architecture is designed to hide this as much as possible from users, as will become apparent.

To explain Mallory's design, we will walk through an entire observe-orient-decide-act loop, step by step, gradually introducing its architectural components.

## 3.1 Observing the System Under Test

Mallory's first task is to observe the system under test (SUT). Broadly, there are three types of observations that we can make: (1) network observations, which capture communication between nodes in the SUT (*e.g.*, a packet was sent from node A to node B and received by node B), (2) external observations, which capture the input-output behaviour of the system (*e.g.*, requests and responses for a database), and (3) internal observations, which capture a node's internal behaviour (*e.g.*, a function was executed, a conditional branch was taken, an error message was logged). In the following, we use "observation" and "event" interchangeably.

Events happen on a particular node at a particular time. However, as is well known, in a distributed system there is no globally shared notion of time. We postpone the explanation of how Mallory constructs a global timeline without assuming precise clock synchronisation and without tagging messages with vector clocks. For now, it suffices to say that each event carries a node identifier and a *monotonic timestamp* returned by the node's system clock.

Below, we outline how Mallory observes the defined above types of events capturing the patterns of communication (Sec. 3.1.1), externally observable input/output (Sec. 3.1.2), and internal behaviour of the nodes (Sec. 3.1.3).

*3.1.1 Packet Interception.* To keep our framework lightweight and require as little modification of the system under test as possible, we capture TCP and UDP packets at the IP network layer using Linux's firewall infrastructure, rather than require users to instrument the application layer to identify protocol-level messages.

By necessity, Mallory's architecture is distributed, matching the structure of the SUT. As shown in Fig. 3, Mallory consists of a number of *observer* processes, one at each node, that observe local events (bottom half of the figure), and a central *mediator* process that collates information from all observers and coordinates the execution of the test (large blue rectangle in the top half). At every node, the observer, which the Jepsen test harness starts before the system under test, installs a Netfilter firewall queue that intercepts all IP packets sent to or from the node. During the test, the kernel copies packets to the observer process in user space, where each packet is assigned a monotonic timestamp, recorded, and then emitted unchanged.

***Mediator interception***. Observers collect packet events in batches and forward those to the mediator periodically, by default every 100ms. Rather than include the entire packet in the batch, which would entail trebling network traffic, observers only record and send to the mediator a 64-bit packet identifier obtained from the source and destination IP addresses and ports and from the IP and UDP or TCP headers' identifiers, respectively. Yet we do want the mediator to have access to the packet contents: for instance, the content of messages might determine what is the best fault to introduce. To achieve this, we set up the test environment that the SUT executes in such that *all packets pass through the node running the mediator*. Concretely, we place each node on its own separate (virtual) Ethernet LAN, with the mediator acting as the gateway for all the LANs. The Mallory mediator acts as a *man-in-the-middle* for all packets in the SUT. It can then reconstruct the identifier for each packet, and cross-reference it with the batches received from the sending and receiving nodes' observers to determine the respective timestamps. The mediator, unlike the observer, which is passive, is active and implements a full user-space firewall using Netfilter. It can delay and drop packets when instructed to do so by the decide step of the OODA loop.

Using this infrastructure, the mediator builds up a complete picture of the system under test's communication.

*3.1.2 Requests and Responses.* The observations about input/output of nodes are made at the application layer.

We built Mallory on top of Jepsen, and reuse Jepsen's infrastructure to define the test harness that: (a) sets up and starts the SUT, (b) defines and executes a workload (a sequence of client requests to the SUT), (c) enacts faults (*e.g.*, crashing a node), and (d) checks the validity of the SUT's response to the workload. Jepsen already captures requests and responses for validity checking (*e.g.*, for linearisability), and we hook into this existing code, attach monotonic timestamps to events, and relay them to the mediator reactively.

*3.1.3 Code Instrumentation.* The final kind of observation we make is at the code level. We want Mallory to be able to peek into the internal workings of the SUT, beyond what is observable to clients of the system or to eavesdroppers on the network. For this, we reuse the compile-time instrumentation infrastructure used by greybox fuzzers (*e.g.*, AFL) for sequential programs. Like those fuzzers, Mallory adds *instrumentation code* to the SUT to capture and expose runtime information about the program's execution. The key question is: what about the execution should we capture?

In our early experiments, we used the notion of *edge coverage*, the type of instrumentation that has become standard for fuzzing sequential programs due to its empirically-observed effectiveness. It maintains a global bitmap of code edges, and increments an approximate counter for each edge that is traversed during program execution. At the end of the execution, the bitmap serves as a summary of "what the program did," and is used by the fuzzer to assign energy and mutate its input during subsequent runs. This is a great metric for certain kinds of programs, *e.g.*, command-line utilities and file-parsing libraries, but—as we quickly discovered—not particularly meaningful for distributed systems. The goal in fuzzing sequential programs is to generate inputs that go "deep" into a program and explore all "cases" (*i.e.*, conditional branches). For such programs, the thoroughness of exploration is naturally defined in terms of code coverage. But this is not the case at all for distributed systems. Distributed systems tend to be implemented as reactive event loops and run almost the same code for every request, with minor variations. Code coverage metrics tend to saturate very quickly when testing such systems.

A natural behavioural metric for distributed systems, which we came to adopt, is that of the *event trace*. Executions in a distributed system are distinguished not so much by which events happen, but by the order in which they happen. Moreover, as has been empirically observed, what tends to uncover bugs are specific subsequences of events, *e.g.*, $A$ before $B$ before $C$, with potentially many events between them [46]. The disadvantage of event traces compared to code coverage is that the former can become very large and expensive to store and operate on, especially if every basic block is instrumented. To alleviate this issue, for now we require from the user a small amount of manual annotation of the SUT's code, in the form of `//INSTRUMENT_FUNC` and `//INSTRUMENT_BLOCK` comments, to indicate which basic blocks and functions are "interesting" and should be tracked by Mallory.

Our instrumentation creates a POSIX shared memory object accessible by the observer process, and stores in it a fixed-size global array of events along with an atomic index. We implement a LLVM pass that assigns a unique ID to every annotated basic block and function in the SUT, and inserts the hooking code at the start of the block or function. During program execution, this code gets a monotonic timestamp and records the event in the global array at a fresh position. The observer process periodically reads the shared memory object, copies the trace, and resets the counter; it also includes the trace in the periodic batch it sends to the mediator.

## 3.2 Making Sense of Observations

For the second phase of the loop, Mallory needs to make sense of the events it received from observers. The goal of this phase is to transform the "raw" event data into a form more amenable to analysis and decision-making.

**Algorithm 2:** Build prefix-closed Lamport timeline

```
Input: Map from node IDs to ordered sets of events
Output: Prefix-closed causal timeline as a graph
(prefixRanges, extensionRanges) ← readyRanges()
links ← trackLinkSources(extensionRanges)
rw ← map(first, prefixRanges) /* resumeWith */
ln ← map(last, prefixRanges) /* lastNeeded */
while $\exists n.\ \mathrm{rw}[n] \neq \mathrm{ln}[n]$ do
    for $n \in$ nodes do
        while $\mathrm{rw}[n] \neq \mathrm{ln}[n]$ do
            range ← $\mathrm{rw}[n]$ .. $\mathrm{ln}[n]$
            for ev ∈ range do
                if ev is the target of src ∈ links then
                    attachLinkTarget(ev, links)
                    ln[node(src)] ← max(src, ln[node(src)])
                rw[$n$] ← ev
Add all events and links up to ln[$n$] to the graph.
```

It is at this stage that our key conceptual contributions of *timeline-driven testing* and *timeline abstraction* come into play. At the core of Mallory's OODA loop lie its *view* of the world, a dynamically constructed Lamport timeline of events in the SUT, and its *model* of the world, a user-defined abstraction of the timeline. Mallory first builds a birds-eye view of the SUT's execution by constructing a global timeline, then makes sense of the timeline by abstracting it into a summary consisting of its "essential" parts, which is used to judge the effectiveness of previous actions and to decide which faults to introduce next.

*3.2.1 Building the Timeline.* As the system is executing, the Mallory mediator receives batches of events from all the observers and adds them into a single global timeline. Every event is associated with a particular node in the SUT and has an attached monotonic timestamp from that node's system clock. However, events *do not* have causal timestamps (*e.g.*, vector clocks) that encode the causal relationship between events at different nodes. In other words, the mediator at this stage has a timeline with events, but no causal arrows between events on different nodes. It must use its complete knowledge of the system's communication to reconstruct the causal ordering of observed events.

Importantly, the timeline that is passed on to the next stage to be abstracted must be *prefix-closed*, in the sense that if an event $e$ is included, all its causal predecessors must also be included. Due to the distributed nature of the system, there are some complications in creating prefix-closed timelines: (a) batches of events from a single observer might arrive out of order and (b) receipt events may arrive before their respective send events (in other words, the target of a causal arrow may arrive before the source). The first issue is straightforward to solve: give a sequence number to each batch, and have the mediator ensure batches are added to the timeline in sequential order. This is sufficient to guarantee that each node's local timeline is prefix-closed. The second issue is more challenging due to the recursive nature of prefix-closedness across nodes. Recall that the mediator now has a timeline with events, but no causal arrows. The question is: how can we know that we have received all the causal predecessors of a given event, *i.e.*, that an event belongs to the prefix-closed portion of the timeline?

There are multiple ways to solve this problem. The approach we choose exploits a well-known property of causality: real-time order is an over-approximation of causal order, *i.e.*, if event $B$ happens in real-time after event $A$, then $B$ cannot causally influence $A$. In other words, all events $A$ that causally influence an event $B$ must be before $B$ in real-time order. This means that we only need to look for an event's causal predecessors in the timeline up to the point where the event's real-time timestamp is first exceeded. The issue is that (c) events are tagged with monotonic, not real-time timestamps, and (d) nodes in any case do not have synchronised real-time clocks. We address (c) by requiring each observer to submit upon start-up both a monotonic timestamp and a real-time timestamp obtained at roughly the same time. This lets us approximately convert monotonic timestamps at a single node into real-time timestamps for that node. The issue (d) of clocks not being synchronised across nodes still remains, however. The solution: we introduce a bound on the maximum clock skew between nodes (we set it to a conservative 100ms), and use this to limit "how far in the future" we look for causal predecessors of an event $e$ on other nodes' timelines, relative to $e$'s real-time timestamp.

Algorithm 2 shows our timeline construction procedure. It takes as input a map from node IDs to the ordered set of events at that node, and constructs a prefix-closed causal timeline (aka a *consistent cut* [31]) in the form of a graph—events are nodes and causal arrows are edges. The algorithm proceeds in four stages. First, on line 1, we identify for each node the range of events that have not yet been processed and whose causal predecessors must all exist in the input (`prefixRanges`). This is obtained by subtracting the maximum clock skew ($cs$) from the last real-time timestamp ($ts$) up to which *all* nodes have submitted events. We are guaranteed to have all the causal predecessors of events in this range—these will be found in the range up to $ts + cs$ on other nodes (`extensionRanges`). Second, on line 2, we traverse the extension ranges, identify all events that are the source of inter-node causal arrows (*i.e.*, packet sends, which, recall, have 64-bit identifiers), and keep track of these causal arrows, identified by the source's unique ID (`links`). The third stage (lines 5–13) is a fixpoint computation that keeps track of where the prefix-closed portion of the timeline ends (`lastNeeded`) for each node and "fills up" the timeline by traversing it (lines 8–9) and "requiring" that causal predecessors of encountered events also be in the timeline (lines 11–12). The fixpoint computation terminates (line 5) when there are no gaps for any nodes, *i.e.*, the timeline is prefix-closed up to $\ln[n]$ for all nodes $n$. The fourth and final stage (line 14) is to construct a graph from the prefix-closed portion of the timeline and pass it to the abstraction phase. (The graph is prefix-closed, *i.e.*, all receive have a matching send, but not necessarily suffix-closed, *i.e.*, there may be sends with no matching receive. The causal arrows for these point to a special node at infinity, and will point to the correct receive when it is encountered later.)

### 3.2.2 Timeline Abstraction.

We now have a Lamport timeline describing the system's behaviour, obtained in almost real-time, and want to use it to drive our testing campaign. More concretely, we want our decisions to adapt the system's behaviour and drive the execution towards new behaviours. But what counts as novel behaviours? A naive approach is to use the timeline itself as feedback for our decision: we want to see timelines different from what we have seen before. This does not work because the timeline is a low-level representation of the system's behaviour; all observed timelines are unique, even discounting event timestamps and packet contents. Clearly, to be able to operate effectively with timelines, we need in some fashion to *abstract* them into "what really matters". Eliminating timestamps and packets is a form of abstraction, but it is not enough: we need something a bit more clever.

```rust
pub trait TimelineAbstraction {
  fn update(&mut self, ev: &Event);
  fn merge(&mut self, this_ev: &Event,
           other: &Self, other_ev: &Event);
}
```

**Figure 4: Rust interface for defining timeline abstractions. Abstractions are built incrementally by iterating over the causal structure, storing "what matters" along the way.**

Importantly, we do not want to bake into the tool any particular notion of "what really matters". (We do provide sensible defaults.) Instead, we want to allow users of Mallory to specify what is important for their particular systems and testing needs. Moreover, we want an intuitive interface for specifying this, one that any distributed system engineer can understand and use. To this end, we introduce our novel notion of *timeline abstraction*, inspired by vector clocks. Our insight is that the principle used to define causal timestamps, namely that of *accumulating causal information* via `update` (aka *event* or *copy*) and `merge` (aka *join*) operations [15, 31], can be generalised to arbitrary abstractions of causal diagrams. The interface for defining such abstractions is shown in Fig. 4. Intuitively, a timeline abstraction is attached to an event and represents or *abstracts* the whole causal history up to (and including) that event. For example, a vector clock attached to an event represents the event's causal "position" in the timeline, allowing for causal comparisons between different events. In a sense, the vector clock—maintained by iterating over the timeline's structure (`update` for same-node events) and (`merge` for inter-node causal dependencies)—compresses the whole timeline up to that event into a single value that captures "what is essential" for the purpose of determining causal relations between events. But what if our goal is different, *e.g.*, to *summarise* what happened in a timeline in order to inform our testing? Fig. 5 shows a simplified version of the default abstraction used by Mallory. Instead of accumulating a causal timestamp like vector clocks do, an `EventHistory` accumulates for every node in the timeline a set of *happens-before pairs* of event types that occurred locally at that node. To get an intuition for this, Fig. 3 shows in its centre portion a timeline with its associated `EventHistory`.[3] The abstraction is obtained by traversing the timeline's events in causal order (starting from an empty `EventHistory`), and calling `update` for events on the same node and `merge` for inter-node causal dependencies. The result is a summary of the timeline's events that captures some of the history's essential aspects and which we use in the next stage to decide which faults to introduce.

[3] More precisely, the shown `EventHistory` is associated with an artificial event that is causally after every node's last event.

```
pub struct EventHistory {
  events: Map<NodeId, Set<EvKind>>,
  pairs: Map<NodeId, Set<(EvKind, EvKind)>>,
}
impl TimelineAbstraction for EventHistory {
  fn update(&mut self, ev: &Event) {
    self.events[ev.node].insert(ev.kind);
    for ev_a in self.events {
      let pair = (ev_a, ev.kind);
      self.pairs[ev.node].insert(pair);
    }
  }
  fn merge(&mut self, other: &Self, ..) {
    /* For every NodeID, take set union. */
  }
}
```

**Figure 5: A timeline abstraction that tracks *happens-before pairs* of event types on a per-node basis, *e.g.*, a `Commit` happened before a `Rollback` at the same node.**

### 3.3 Making Decisions with Q-Learning

Equipped with a way to understand the behaviour of the SUT, in the form of timeline abstractions, Mallory must decide which actions to take in response to what it observes.

For fuzzing sequential programs, *mutation-based power scheduling* has become the standard approach to generate novel inputs for the program under test based on observations: test inputs that exercise new behaviours are stored and mutated many times to obtain new inputs. However, this technique is ill-suited for testing distributed or reactive systems. The issue is that in the mutation-based paradigm, behavioural feedback is given for the whole input to the system under test (SUT) after execution ends. This is reasonable for sequential programs, but not for reactive programs—the (temporal and causal) connection between fault introduced and behaviour induced is lost. Indeed, we want our fuzzer itself to be reactive and give behavioural feedback after every action taken rather than only at the end of a long schedule. This complements Jepsen's *generative fuzzing* approach by giving behavioural feedback after every generated fault and makes Mallory adapt in real-time to the SUT.

The concept of *timeline-driven testing* is key to how Mallory adapts to the SUT. To decide which faults to introduce into the SUT, Mallory employs Q-learning [42, 43], a model-free reinforcement learning approach. Q-learning enables an agent to dynamically learn a *state-action policy*, *i.e.*, a pairing between the observed state of the environment and the optimal action to take. Based on the observed states, this policy guides the agent in selecting actions to take. After performing an action, the agent receives an immediate reward, which further refines the policy. By associating observed states with optimal actions, the agent maximizes the expected rewards over its lifetime.

Q-learning can be seamlessly integrated into our distributed system fuzzer. With this approach, our fuzzer (*i.e.*, the agent) learns a policy throughout the fuzzing campaign, which serves as a guide to explore diverse states. Specifically, during the fuzz campaign, our fuzzer observes the state $s_i \in \mathcal{S}$ of the SUT, and then employs the learned policy to select a fault $a_i \in \mathcal{A}$ (*i.e.*, an action) to introduce, causing it to transition to the next state $s_{i+1}$. Simultaneously, the fuzzer receives an immediate reward $r_i \leftarrow \mathcal{R}(s_i, a_i, s_{i+1})$. Using the received reward, our fuzzer further refines the policy $\pi : \mathcal{S} \mapsto \mathcal{A}$. Subsequently, our fuzzer selects the next fault $\pi(s)$ to introduce. Over time, our fuzzer progressively learns an optimal policy that maximizes the number of distinct states observed (*i.e.*, rewards). In the following, we elaborate this process further.

**Capturing States**. In the context of Q-learning, states should represent the behaviour of the environment. As elaborated in Sec. 3.2, to describe the behaviour of the SUT, we adopt the timeline abstractions (*e.g.*, a set of happens-before pairs). To incorporate Q-learning into our framework, we take a hash of the timeline abstractions to serve as *abstract states*. However, while computing states, we encounter a challenge due to the sensitivity and precision of our timeline abstractions. In order to improve the learning speed of the state-action policy, we aim to treat states that differ only insignificantly as identical. Using a standard hash function for this purpose proves ineffective, as it tends to be overly sensitive to minor variations in the timeline abstraction. Such variations can arise even when the system is operating without any injected faults and under a constant load, owing to the inherent non-deterministic nature of distributed systems.

To address this issue, we adopt MinHash, a locality-sensitive hash function that maps similar input values to similar hash values. In our specific case, to decide whether an observed timeline abstraction corresponds to a new distinct state, we hash it into a signature and then compare this signature to those of previously encountered states. We classify a state as distinct if the similarity falls below a threshold $\varepsilon$. To choose $\varepsilon$, we conduct a calibration stage before fuzzing by running the SUT without any faults and under constant load and observing the timeline abstractions thus obtained. The convention in previous fuzzing works, which we also adopt, is to choose an $\varepsilon$ value that makes 90% of such "steady" states coincide [30, 33].

**Learning the Policy**. In our approach, the policy $\pi$ is represented as a Q-table, where each column corresponds to a specific type of fault, and each row represents a distinct state. The available fault types are provided by the fuzzer and enabled at the start of the fuzzing campaign. As new distinct states are observed and added, the rows dynamically increase to accommodate them. Within the Q-table, each cell stores the Q-value for a state-action pair. When a new distinct state is added to the table, the Q-values for each state-action pair associated with that state are initialized to 0. With this setup, the process of refining the policy becomes simplified, involving adjustments to the Q-values.

Q-values are updated in response to rewards. After executing a fault, our fuzzer receives an immediate reward determined by the reward function. The reward function is devised based on our goal. Since our goal is to maximise the number of distinct states observed, we set our reward function to give a constant negative reward (-1) to states that have been observed previously. This approach incentivizes Mallory to steer the SUT towards unobserved behaviors, promoting exploration. Specifically, we design the reward function as follows:

$$\mathcal{R}(s_i, a_i, s_{i+1}) = \begin{cases} -1, & \text{if } s_{i+1} \in \mathcal{S}_{observed} \\ 0, & \text{if } s_{i+1} \notin \mathcal{S}_{observed} \end{cases} \quad (1)$$

Once receiving the rewards, our fuzzer dynamically adjusts Q-values using the Q-learning function $\mathcal{Q}: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, which determines the Q-value for a specific state-action pair. When an action $a_i$ is executed, it leads to a new state $s_{i+1}$ from the previous state $s_i$ (*i.e.*, $s_i \xrightarrow{a} s_{i+1}$). Subsequently, we update the Q-value as follows:

$$\mathcal{Q}(s_i, a_i) \leftarrow (1-\alpha)\mathcal{Q}(s_i, a_i) + \alpha\left(\mathcal{R} + \gamma \max_{a'} \mathcal{Q}(s_{i+1}, a')\right) \quad (2)$$

where $\alpha \in (0, 1]$ indicates the learning rate and $\gamma \in (0, 1]$ is a discount factor. The default values chosen for these parameters are $\alpha = 0.1$ and $\gamma = 0.6$, which we determined to work well empirically. With the Q-function, the new Q-value is computed and subsequently updated into the Q-table. As the fuzz campaign proceeds, our fuzzer gradually fine-tunes the Q-values, thus refining the policy to make better decisions.

**Getting Next Fault**. When the SUT is in a state $s$, our fuzzer selects the next action $a$ based on the learned policy. In the current state $s$, we first obtain the Q-values for all $n$ actions and then utilize the softmax function to convert these Q-values into a probability distribution $\mathcal{D}$. The probability $\mathcal{D}(i)$ for each action $a_i$ $(i \in n)$ is calculated as follows:

$$\mathcal{D}(i) \leftarrow \frac{e^{\mathcal{Q}(s, a_i)}}{\sum_{j=1}^{n} e^{\mathcal{Q}(s, a_j)}} \quad (3)$$

To decide the next action to take, we sample from the probability distribution $\mathcal{D}$. To achieve this, we generate a random number $p \in [0, 1]$, and then check the cumulative probability of $\mathcal{D}(i)$ until we identify the first action where the cumulative probability exceeds $p$. This action is chosen as the next action to execute. Actions with higher probabilities have a greater likelihood of being chosen, while actions with lower probabilities still have a chance of being selected. This approach ensures a balance between favoring actions with higher probabilities while maintaining the possibility of choosing actions with lower probabilities.

Thus far, we have introduced each individual component of Algorithm 1. To kickstart the fuzzing campaign, we set the default maximum steps as 12, each with a 2.5-second window. Following each schedule, there is a 5-second period allocated to reset the system to a stable state.

## 4 EVALUATION

We implement Mallory on top of Jepsen-0.2.7, to test distributed system implementations written in C, C++, and Rust. To enable code-level instrumentation, we created a LLVM compiler pass (similar to that used by AFL [48]) to add into the compiled binary our event instrumentation, as described in Sec. 3.1.3. The code implementing this pass measures roughly 1,000 lines of C/C++ code. The observers at each node that collect events and the mediator which collates events from all observers, intercepts packets, constructs and abstracts timelines, and learns the policy required to guide fault injection, are implemented in Rust. The code for these components measures roughly 9,000 lines of Rust code. To enact faults, we implemented a linker in Jepsen that asks Mallory for the next fault to execute. This linker consists of 140 lines of Clojure code.

**Table 1: Detailed information about our subject programs.**

| Subject | Description | Protocol | Lang. | #LOC | #Stars |
|---|---|---|---|---|---|
| Braft | Baidu Raft implementation | Raft | C++ | 31.6k | 3.5k |
| Dqlite | Distributed SQL DBMS | Raft | C | 54.2k | 3.4k |
| MongoDB | Distributed NoSQL DBMS | Raft | C++ | 1121.6k | 23.6k |
| Redis | Distributed in-memory DBMS | Raft | C | 211.4k | 59.6k |
| ScyllaDB | Distributed NoSQL DBMS | Raft/Paxos | C++ | 122.4k | 9.8k |
| TiKV | Distributed key-value DBMS | Raft | Rust | 404.5k | 13.0k |

### 4.1 Evaluation Setup

To evaluate the effectiveness and efficiency of Mallory in exploring distinct program behaviours and finding bugs in industrial distributed system implementations, we have designed experiments to address the following questions:

**RQ1 Coverage achieved by Mallory.** Can Mallory cover more distinct system states than Jepsen?

**RQ2 Efficiency of bug finding.** Can Mallory find bugs more efficiently than Jepsen?

**RQ3 Discovering new bugs.** Can Mallory discover new bugs in rigorously-tested distributed system implementations?

*4.1.1 Baseline tool.* We selected Jepsen as our baseline tool due to its popularity in stress-testing distributed system implementations. To our knowledge, Jepsen is the only widely-used black-box fuzzer in this domain. It has gained recognition for its user-friendliness and has helped to uncover numerous bugs in real-world implementations of distributed systems. By building on top of Jepsen, we have developed Mallory to enhance the effectiveness of fuzzing while preserving Jepsen's ease of use. Another black-box fuzzer, called Namazu [1], is less popular and can only test Go/Java programs.

As described in the introduction, white-box fuzzers such as MoDist [45] and FlyMC [28] require an extensive manually-written test harness or heavy deterministic control at the system level, and are used for systematic testing as opposed to stress-testing. Due to their heavy-weight nature, they target a different use case compared to Mallory and are less practical to adopt in industry.

*4.1.2 Subject programs.* Tab. 1 presents the subject programs included in our evaluation. It consists of six open-source distributed system implementations written in C, C++, and Rust. We selected these subjects because: (1) they are widely used in the industry, (2) they can be instrumented by our LLVM pass, and (3) they have undergone rigorous testing using Jepsen either by contracting Jepsen's author[4] or by rolling their own Jepsen test harness. Finding new bugs in these systems would be a strong indication that Mallory performs better than Jepsen.

*4.1.3 Event Annotations.* To instrument code events, we annotated a total of 103 to 157 functions and basic blocks in our subject programs using `//INSTRUMENT_FUNC` and `//INSTRUMENT_BLOCK` forms. Specifically, we made 120 annotations for Braft, 108 annotations for Dqlite, 103 annotations for MongoDB, 129 annotations for Redis, 157 annotations for ScyllaDB, and 138 annotations for TiKV. This process required one of our authors to dedicate a total of 2 hours, averaging approximately 20 minutes for each subject.

[4] Test reports are public at https://jepsen.io/analyses

```
145  int membershipRollback(struct raft *r){
146    ...
158    // Fetch the last committed configuration entry
159    entry = logGet(&r->log, r->config_index);
160    assert(entry != NULL);
176  }

985  // INSTRUMENT_FUNC
986  static int deleteConflictingEntries(){
987    ...
1007   // Possibly discard uncommitted config changes
1008   if (uncommitted_config_index >= entry_index){
1009     // INSTRUMENT_BLOCK
1010     rv = membershipRollback(r);
1011   }
1043 }
```

**Figure 6: Annotating Dqlite for membership rollback.**

We annotate “interesting" code events primarily based on the Raft and Paxos TLA+ specifications.[5] We now demonstrate the annotation process for events in the motivation example (Fig. 1), and present the annotated code in Fig. 6. In the Raft TLA+ specification, one event involves the removal of conflict entries. To annotate this event, we search for the “conflict" keyword in the Dqlite source code, resulting in 7 matched locations. However, among these matches, only one location pertains to functions or basic blocks, specifically the “deleteConflictingEntries" function. Therefore, we add the annotation `//INSTRUMENT_FUNC` before this function. Instead of annotating a function - we can also decide to annotate (some of) the basic blocks representing the calling locations of the function. However, this may require additional work in terms of determining which calling location to instrument. For example, for the "membershipRollback" function, we can annotate the basic block in line 1009 that invokes this function with the annotation `//INSTRUMENT_BLOCK` shown in Fig. 6. But choosing this calling location, takes into consideration the condition (line 1008) that there must be an uncommitted configuration entry among the conflicting entries to be removed (explained in Sec. 2.1). Once the code is annotated, our compile-time instrumentation automatically instruments the annotated code without any manual effort.

**Discussion**. It is worth mentioning that we adopt the heuristic based on TLA+ specifications to instrument events in our evaluation. Nevertheless, users of Mallory have the flexibility to instrument their own custom event types in accordance with their own heuristics. For instance, this might involve the instrumentation of error handlers [46] or *enum* cases [4]. The identification of such events can be achieved fully automatically, and we have integrated this functionality into the Mallory tool.

*4.1.4 Configuration parameters.* To distinguish between distinct states, we set the similarity threshold $\varepsilon$ to 0.70. To detect bugs, we adopt several test oracles: (1) AddressSanitizer (*i.e.*, ASan) for exposing memory issues, (2) log checker to detect issues in application logs by scanning for keywords such as “*fatal*”, “*error*” and “*bug*”, and (3) consistency checker Elle [22] to check consistency violations. We set up the same number of nodes as the existing Jepsen tests: 9 nodes for MongoDB and 5 nodes for the other subjects under test. To ensure a fair comparison, we enabled the same faults in Mallory as those used in the original Jepsen tests (*i.e.*, our tool does not have access to more fault types).

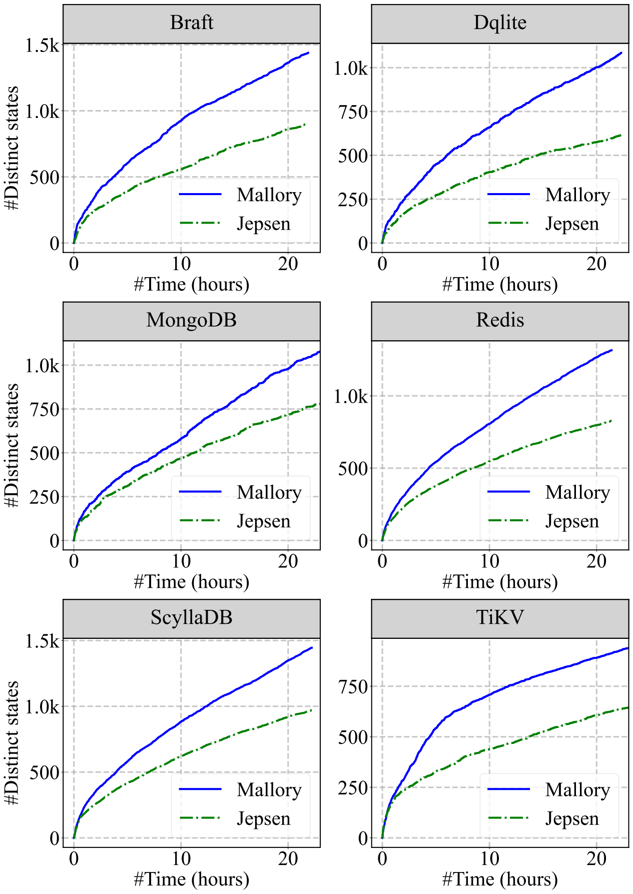

**Figure 7: The trends in the average number of distinct states within 24 hours across 10 runs.**

All experiments were conducted on Amazon Web Services using the m6a.4xlarge instance type. This instance type has 64 GB RAM and 16 vCPUs running a 64-bit Ubuntu TLS 20.04 operating system. Following community suggestions, we ran each tool for 24 hours and repeated each experiment 10 times. To reduce statistical errors, we report as results the average values obtained over the 10 runs.

## 4.2 Coverage Achieved by Mallory (RQ1)

For the first experiment, we monitor the number of distinct states (see Sec. 3.3 for the definition of states) exercised by Mallory and Jepsen over time, and compare their state coverage capabilities.

To observe states exercised by Jepsen, we ran Jepsen in the same setup as Mallory, but without controlling the fault injection. We collected the average number of distinct states achieved by Mallory and Jepsen within 24 hours across 10 runs, and we present the comparison in Fig. 7. As shown in this figure, Mallory outperforms Jepsen by covering more distinct states in the same time budget, thus exercising the SUT under more diverse scenarios.

[5] The specifications are taken from https://github.com/tlaplus/Examples.

**Table 2: Statistics of distinct state numbers achieved by MALLORY compared to that achieved by JEPSEN.**

| Subject | State-impr | Speed-up | $\hat{A}_{12}$ | $U$ |
|---|---|---|---|---|
| Braft | 59.34% | 2.28× | 1.00 | <0.01 |
| Dqlite | 76.14% | 2.56× | 1.00 | <0.01 |
| MongoDB | 36.48% | 1.57× | 1.00 | <0.01 |
| Redis | 58.92% | 2.06× | 1.00 | <0.01 |
| ScyllaDB | 48.82% | 1.88× | 1.00 | <0.01 |
| TiKV | 45.93% | 3.07× | 1.00 | <0.01 |
| **AVG** | 54.27% | 2.24× | - | - |

Initially, at the start of each experiment, the number of distinct states achieved by MALLORY is similar to that achieved by JEPSEN, which is expected. MALLORY utilizes Q-learning to learn an optimal state-action policy during fuzzing, guiding it to make better decisions about actions to take in observed states. However, during the initial phase of fuzzing, this policy is not yet well-learned and lacks sufficient knowledge to avoid exploring redundant states. Consequently, both MALLORY and JEPSEN struggle to select the optimal actions for the observed states.

However, as fuzzing progresses, MALLORY gradually refines the policy by assigning negative rewards to certain state-action pairs, thereby disincentivizing certain actions selected in particular states. This disincentive effectively curtails the exploration of repetitive states and steers MALLORY towards exploring more diverse states. The policy learned using Q-learning proves to be highly effective. As evident in Fig. 7, over time, the number of distinct states covered by MALLORY is significantly more than that covered by JEPSEN. We do not observe the number of distinct states saturating in either tool, but MALLORY's exploration rate of distinct states is higher than that of JEPSEN. Additionally, the gap between MALLORY and JEPSEN consistently widens, indicating that the Q-learning policy continuously improves and becomes increasingly effective in guiding MALLORY's exploration process.

The state coverage statistics of MALLORY over JEPSEN are listed in Tab. 2. The "State-impr" column shows the average improvement of MALLORY in the number of distinct states at the end of 24 hours, over 10 runs. Our results show that MALLORY covers an average of 54.27% more distinct states than JEPSEN on our test subjects, with an improvement ranging from 36.48% to 76.14%. The "Speed-up" column indicates the average speed-up of MALLORY compared to JEPSEN in achieving the same number of observed states. On average, MALLORY archives a 2.24× speed-up over JEPSEN. To mitigate the effect of randomness, we measured the Vargha-Delaney ($\hat{A}_{12}$) and Wilcoxon rank-sum test ($U$) of MALLORY against JEPSEN. $\hat{A}_{12}$ is a non-parametric measure of effect size that provides the probability that random testing of MALLORY is better than random testing of JEPSEN. $U$ is a non-parametric statistical hypothesis test that determines whether the number of distinct states differs across MALLORY and JEPSEN. We reject the null hypothesis if $U < 0.05$, indicating that MALLORY outperforms JEPSEN with statistical significance. For all subjects, $\hat{A}_{12} = 1$ and $U < 0.01$ for MALLORY against JEPSEN. This demonstrates that MALLORY significantly outperforms JEPSEN.

Furthermore, we measured the memory consumption required to maintain the data structure of the Lamport-style timeline. The average memory consumption was 3.21 GB, which we consider acceptable. Memory consumption remains stable over time, as we only retain the portion of the timeline needed for abstraction and remove already-abstracted portions. Additionally, our fuzzer is designed to learn and react to observations in real time (see Sec. 3.3). We measured the time taken from the point of fault injection to receiving the behaviour feedback and found that in 92.20% of cases, this process took less than one window time, *i.e.*, MALLORY receives feedback and adapts its policy before it has to decide the next action.

> In terms of state exploration, MALLORY covers 54.27% more distinct states than JEPSEN with a 2.24× speed-up.

## 4.3 Efficiency of Bug Finding (RQ2)

To evaluate the efficiency of MALLORY at finding bugs, we compared MALLORY and JEPSEN with regards to the time required to reproduce existing bugs. To this end, we created a dataset of bugs by selecting 10 recent issues from each subject's GitHub issue list (from early 2019 to April 2023) that contained instructions for reproduction. We attempted to reproduce the bugs manually and included any successfully reproduced bugs in our dataset. We finally collected a total of 16 bugs across all subjects. The bug IDs and types of bugs are presented in the first two columns of Tab. 3.

We ran both tools, MALLORY and JEPSEN, on buggy versions of the subjects for 24 hours, repeated 10 times. The last main column shows the time used for each tool to expose the bug. We marked "T/O" if one tool failed to find the bug within the given time budget. Overall, in these 16 bugs, MALLORY successfully exposed 14 bugs, while JEPSEN only found 9 bugs. In terms of time usage, MALLORY takes much less time (*i.e.*, 6.13 hours on average), while JEPSEN needs 11.45 hours. Hence, compared with JEPSEN, MALLORY achieves a speed-up of 1.87× in bug finding.

For shallow bugs whose states are easy to reach, such as Dqlite-338 and Dqlite-327, MALLORY and JEPSEN perform well and perform similarly. However, for deep bugs that are harder to expose, MALLORY performs much better than JEPSEN. For example, to expose Redis-51, JEPSEN took 6.40 hours, while MALLORY only took 1.66 hours. This is attributed to a faster state-exploration speed of MALLORY. In addition, since MALLORY explored more distinct states than JEPSEN, MALLORY was also able to expose more bugs. Specifically, MALLORY successfully exposed Dqlite-324, Dqlite-323, Redis-28, Redis-23, and Redis-17, while JEPSEN had difficulty in exposing them. We further investigated the two bugs (*i.e.*, Dqlite-356 and Dqlite-314) missed by MALLORY, and found exposing these bugs needs to inject specific environment faults (*e.g.*, disk faults), which were not included in our evaluation.

To mitigate randomness, we adopt the Vargha-Delaney ($\hat{A}_{12}$) to measure the statistical significance of performance gain. The last subcolumn of Tab. 3 shows these results. We mark $\hat{A}_{12}$ values in bold if they are statistically significant (taking 0.6 as a significant level, or threshold). We can see that, in most cases, MALLORY significantly outperformed JEPSEN.

> In terms of bug finding, MALLORY finds 5 more bugs and finds bugs 1.87× faster than JEPSEN.

**Table 3: Statistics of reproduced known bugs and the performance of both Mallory and Jepsen in exposing these bugs.**

| Bug ID | Type of bug | Time to exposure | | |
|---|---|---|---|---|
| | | Mallory | Jepsen | $\hat{A}_{12}$ |
| Dqlite-416 | Null pointer deference | **0.76h** | 1.44h | **1.00** |
| Dqlite-356 | Snapshot installing failure | T/O | T/O | 0.50 |
| Dqlite-338 | Election fatal with split votes | 0.16h | 0.16h | 0.50 |
| Dqlite-327 | Member removal failure | 0.06h | 0.05h | 0.49 |
| Dqlite-324 | Log truncation failure | **5.94h** | T/O | **1.00** |
| Dqlite-323 | Membership rollback failure | **8.68h** | T/O | **1.00** |
| Dqlite-314 | Crashing on disk failure | T/O | T/O | 0.50 |
| Redis-54 | Snapshot panic | **3.33h** | 5.00h | **0.95** |
| Redis-53 | Committed entry conflicting | **0.87h** | 1.17h | **0.89** |
| Redis-51 | Not handling unknown node | **1.66h** | 6.40h | **1.00** |
| Redis-44 | Loss of committed write logs | **0.34h** | 0.58h | **0.60** |
| Redis-43 | Snapshot index mismatch | 0.16h | 0.16h | 0.50 |
| Redis-42 | Snapshot rollback failure | 0.29h | 0.26h | 0.50 |
| Redis-28 | Split brain after node removal | **9.56h** | T/O | **1.00** |
| Redis-23 | Aborted read with no leader | **7.29h** | T/O | **1.00** |
| Redis-17 | Split brain and update loss | **11.06h** | T/O | **1.00** |
| Bugs exposed in total | | 14 | 9 | - |
| Average time usage | | 6.13h | 11.45h | - |
| Speed-up on time usage | | - | 1.87× | - |

[1] T/O means that the tool cannot expose bugs within 24 hours for 10 experimental runs. We replace T/O with 24 hours when calculating average usage time.
[2] Statistically significant values of $\hat{A}_{12}$ are shown in bold.

## 4.4 Capability of Exposing New Bugs (RQ3)

To evaluate Mallory's capability of exposing new bugs, we utilised Mallory on the latest versions of our subjects. In the course of the experiment, Mallory produced promising results, as demonstrated in Tab. 4. Although all of these subjects have been rigorously tested by Jepsen and other tests, Mallory was still able to find a total of 22 previously unknown bugs, and 18 bugs of them were confirmed by their developers. Out of these 22 bugs, 10 bugs were associated with vulnerabilities, and we have requested CVE IDs for them. As of the paper submission, we have already obtained 6 CVE IDs and the remaining requests are still being processed.

We conducted a thorough analysis of the nature of these new bugs found by Mallory, shown in Tab. 4. The table also includes the bug checkers used to uncover these bugs. Among these 22 bugs, 7 bugs were determined to be consistency violations exposed by the Elle consistency checker. Three bugs (#3, #4, and #17) violated the Raft protocol due to missing leaders or the existence of two leaders in the same term, and they were detected by the log checker. AddressSantizer (*i.e.*, ASan) exposed 5 memory issues. Furthermore, the log checker detected 7 runtime failures or invariant violations. These results indicate that Mallory is beneficial to expose diverse types of previously unknown bugs.

In addition, we applied Jepsen to detect these 22 new bugs; however, under the allotted time limit, Jepsen was only able to detect four of them (*i.e.*, the bugs #1, #7, #16, and #22), as shown in the last column of Tab. 4. This result is expected because these subject systems routinely undergo Jepsen testing by their developers, making it challenging for Jepsen to discover new bugs.

In the following, we provide two case studies to illustrate bugs that were exposed by Mallory.

***Case study: Bug #2 in Braft***. Braft is a robust Raft implementation designed for industrial applications, which is widely used within Baidu to construct highly available distributed systems. However, a critical vulnerability, known as Bug #2, remained undetected in all Braft release versions from 2019 until its recent patch. This bug occurs when a server cannot release its allocated memory before failure, resulting in a memory leak issue.

To trigger this issue, a minimum of three environmental faults must be introduced sequentially. Initially, the server dynamically allocates enough memory for its operation, which is explicitly managed by itself. However, before releasing the allocated memory, the server is paused, and its memory remains in use. Subsequently, the server is resumed, only to become coincidentally isolated from the cluster due to a network partition, resulting in a failed start. Hence, the server crashes without the chance to release the memory allocated. This bug happens due to a flawed logic design; specifically, the allocated memory can only be released when the process is in the running status, and the allocated memory cannot be released before running. This logic design is reasonable in stable environments without faults, as only the running server may have allocated memory. However, in this extreme environment, the shortcoming in the logic is exposed.

Bug #8 of Dqlite is another memory leak issue, similar to Bug #2 in Braft. It remained hidden in Dqlite for approximately four years and affected all its release versions before we found it. Bug #2 in Braft and Bug #8 in Dqlite both evaded the rigorous testing efforts by their developers, demonstrating how our tool Mallory can significantly reduce the exposure of systems to vulnerabilities.

***Case study: Bug #11 in Dqlite***. Although Jepsen is already part of Dqlite's Continuous Integration process, Mallory has managed to expose several new bugs in Dqlite. The developers have expressed a keen interest in Mallory and are awaiting its open-source release so that they can incorporate it into their testing.

Bug #11 in Dqlite is caused by the snapshotting of uncommitted logs, and it is reminiscent of the membership rollback bug shown in Fig. 2. The schedules required to trigger these bugs are quite similar, but Bug #11 is not triggered by the configuration change. Specifically, the event ① involves one plain read/write log entry instead of the configuration change. After this event, the cluster undergoes the same sequence of environment faults, including a network partition ($\#\{S_1, S_2\}, \#\{S_3, S_4, S_5\}$), leader $S_1$ crashing, and network healing. As a result of these faults, server $S_2$ ends up with conflicting entries with the leader $S_3$, which must be removed. However, the conflicting entries are in a snapshot, which makes the removal fail. This failure leads to the server becoming unavailable. Although the schedule to trigger Bug #11 is slightly shorter than that of the membership rollback bug in Fig. 2, Jepsen, which adopts a random search strategy, failed to expose it during our experiments within the allotted time. In contrast, Mallory, guided by the policy learned with Q-learning, successfully exposed this bug. This is easily explained considering the number of states explored by Jepsen and Mallory shown in Fig. 7. Within 24 hours, Jepsen only covered 600 states, while Mallory explored over 1,000 states. With the capability to explore more novel behaviors, Mallory significantly

**Table 4: Statistics of the zero-day bugs discovered by MALLORY in rigorously tested systems; a total of 22 previously unknown bugs, 18 bugs confirmed by their developers, and 10 software vulnerabilities.**

| ID | Subject | Bug description | Bug checker | Bug status | JEPSEN? |
|---|---|---|---|---|---|
| 1 | Braft | Read stale data after a newly written update is visible to others | ELLE | Investigating | ✓ |
| 2 | Braft | Leak memory of the server when killed before its status becomes running | ASan | CVE-Granted, fixed | ✗ |
| 3 | Dqlite | Two leaders are elected at the same term due to split votes | Log checker | Confirmed | ✗ |
| 4 | Dqlite | No leader is elected in a healthy cluster with an even number of nodes | Log checker | Confirmed, fixed | ✗ |
| 5 | Dqlite | A node reads dirty data that is modified but not committed by another node | ELLE | Confirmed | ✗ |
| 6 | Dqlite | Lose write updates due to split brain | ELLE | Confirmed | ✗ |
| 7 | Dqlite | A null pointer is dereferenced due to missing the pending configuration | ASan | CVE-Requested | ✓ |
| 8 | Dqlite | Leak allocated memory when failing to extend entries | ASan | CVE-Requested, fixed | ✗ |
| 9 | Dqlite | Buffer overflow happens while restoring a snapshot | ASan | CVE-Requested | ✗ |
| 10 | Dqlite | A node has an extra online spare | Log checker | Confirmed | ✗ |
| 11 | Dqlite | Violate invariant as a segment cannot open while truncating inconsistent logs | Log checker | CVE-Requested | ✗ |
| 12 | MongoDB | Not repeatable read due to missing the local write update | ELLE | Confirmed | ✗ |
| 13 | MongoDB | Not read committed due to missing the newly written update | ELLE | Confirmed | ✗ |
| 14 | Redis | Read stale data after new data is written to the same key | ELLE | Confirmed | ✗ |
| 15 | Redis | Buffer overflow due to writing data to a wrong data structure | ASan | CVE-Granted, fixed | ✗ |
| 16 | Redis | Runtime panic on initializing a cluster due to database version mismatch | Log checker | CVE-Granted | ✓ |
| 17 | TiKV | No leader is elected for a long time in a healthy cluster | Log checker | Investigating | ✗ |
| 18 | TiKV | Lose write updates due to split brain | ELLE | Investigating | ✗ |
| 19 | TiKV | Runtime fatal error when one server cannot get context before the deadline | Log checker | CVE-Granted | ✗ |
| 20 | TiKV | Runtime fatal error in a server when the placement driver is killed | Log checker | CVE-Granted | ✗ |
| 21 | TiKV | Runtime fatal error when failing to update max timestamp for the region | Log checker | CVE-Granted | ✗ |
| 22 | TiKV | Monotonic time jumps back at runtime | Log checker | Investigating | ✓ |

increased its chances of exposing bugs. This instance demonstrates the effectiveness of the policy learned using Q-learning in guiding MALLORY to explore more behaviors and, consequently, enhance the likelihood of bug exposure.

> MALLORY found 22 zero-day bugs in rigorously tested implementations, and 18 bugs out of them have been confirmed by their developers. 10 of these 18 bugs correspond to security vulnerabilities, and out of these 6 CVEs have been assigned.

## 5 RELATED WORK

***General Greybox fuzzing***. The vast majority of existing grey-box fuzzers aim at testing sequential software systems, with most of the recent research efforts dedicated to generating more diverse inputs [14, 40], defining better feedback functions [5, 39] and test oracles [32]. With this mindset, fuzzing distributed systems poses unique challenges since (*i*) the inputs include not just plain data but also schedules consisting of environmental faults and (*ii*) code coverage is not as efficient, since distributed systems typically do not have complex control flow and their behavioural complexity comes from the asynchrony of operations across multiple nodes. The recent MUZZ [11] framework for fuzzing of (single-node) multi-threaded programs, addresses (*i*)-(*ii*) by extending the edge coverage metric with possible thread interleavings, while also identifying equivalent schedules. MUZZ' approach does not extend to distributed systems as it is tailored to tracking the ordering of specific threading functions, while MALLORY works with arbitrary events and communication patterns. Furthermore, MUZZ relies on instrumenting the system scheduler, which is difficult to implement for a distributed system without virtualising the networking layer. Moreover, our timeline-based approach offers a more general way to observe program behaviours for *any* non-sequential program, including stand-alone multithreaded as well as distributed systems.

***Greybox fuzzing of protocols***. Recently, greybox fuzzing has been extended to stateful reactive systems. One of the key challenges in this research is state identification - since there are many program variables in an implementation, it is hard to gauge the state variables. AFLNET [38] is one of the early works in this domain. It uses response code as a proxy for states, and mutates sequences of protocol messages based on coverage and other feedback. However, response codes may not be an accurate proxy for stateful behavior. Among other works, STATEAFL [33] utilizes the program's in-memory state to represent the service state. IJON [3] provides motivational examples to show that for some examples it may be intuitive to provide manual annotations to guide fuzzing of stateful software systems. SGFUZZ [4] does not need manual annotations, and instead of simple heuristics to identify state variables such as postulating that state variables are often captured by *enum* type variables. Our work on distributed system fuzzing does not make any such assumptions about state variables. It captures the events executed so far in a reactive system via a timeline abstraction. It also suggests for the first time, the fuzzer of stateful reactive systems, itself as a reactive system. Thus the fuzzer feedback instead of being given for an entire schedule or execution, is given incrementally action by action, with the probabilities for the actions being adjusted via Q-learning.

***Testing***. The majority of state-of-the-art testing frameworks that explore behaviours of a distributed system assume *full control* over the inherent non-determinism of runtime executions, with Jepsen being a notable exception [23]. They achieve this by either (a) replacing the networking layer [45], (b) explicitly modifying SUT to include a test harness [27, 35, 49], or (c) implementing the system in a testing-friendly language [12, 21, 47]. Controlling the asynchrony makes it possible to employ techniques from software model checking such as *partial order reduction* [20] to avoid redundancy when exhaustively exploring the space of bounded runtime executions [13, 28, 35, 36]. While these approaches allow for more effective behaviour exploration than Jepsen/Mallory, they are far more difficult to apply, requiring, correspondingly, (a) specific OS setup, (b) protocol-aware SUT modifications, or (c) using a domain-specific language.

***Model Checking***. Of the existing model-checking frameworks, MoDist [45] is the most similar to our approach, as it requires no modification in SUT code; instead, it manipulates the system's execution by intercepting calls to the Windows API. The main conceptual distinction of MoDist from our work is its interposition layer: unlike our observers, which are passive, MoDist intercepts network, timing, and disk-related system calls and *pauses* the SUT. This provides more control—in our terminology, it allows the mediator to act as a *scheduler*—but comes at the expense of requiring a complex interposition framework that replicates and replaces most of the API of a specific OS. The design innovations of Mallory that enable summarising observations about SUT (Sec. 3.2) are orthogonal to the use of an interposition layer, and, therefore, such an interposition layer could be integrated within our architecture—rather than choosing only which faults to introduce, our nemesis would choose every action. In this work, we focused on finding bugs in non-Byzantine fault-tolerant distributed systems [26] thus, side-stepping the challenge of modelling the behaviour of possibly malicious nodes. We believe that Mallory's workflow can be combined with existing techniques for Byzantine system testing that emulate attacks by running several copies of the same node, but, for now, only allow for execution in a network emulator [6].

***Deductive verification***. Finally one can employ a mix of algorithmic and deductive logical reasoning, albeit not on actual protocol implementations. These approaches broadly fall into one of the following two lines of work. The first line of work is concerned with sound verification of *abstract models* of distributed protocols [7, 25] for safety and liveness properties and focuses on methods for automated reasoning, such as checking and inferring protocol *invariants* (e.g. [37]). Even though some of those approaches allow one to generate executable code from verified protocol models (e.g. [41]), the implementations tend to evolve over time, losing their correspondence to the formally verified models and, hence, relying on testing for correctness. The second line of work concerns verification of *executable* code and typically requires the system to be implemented in a domain-specific language that allows for machine-assisted formal reasoning (e.g. [17]). Such verified implementations incur very high maintenance costs [44] and still remain prone to bugs due to occasional inadequate *assumptions* about the networking infrastructure or third-party libraries [16].

# 6 CONCLUDING REMARKS

In this work we proposed Mallory—the first adaptive greybox fuzzer for distributed systems. The key insight behind Mallory's design is to summarise the runtime behaviour of the distributed system under test in the form of Lamport-style *timelines* that capture causality of events, and use the timelines to define a feedback function for guiding the search for bugs.

Our conceptual contribution of *timeline-driven testing* opens new avenues for automated testing of distributed systems, similar to what was achieved for sequential programs by tools like AFLFast [10]. AFLFast achieves high behavioural diversity by making smart online decisions about covered program paths, during the fuzz campaign. Similarly, Mallory achieves behavioural diversity by making online decisions to detect and prioritise novel event sequences that have not been observed before.

We evaluated Mallory on six widely-used and rigorously-tested industrial distributed system implementations such as Dqlite and Redis. The experimental results show the effectiveness and efficiency of Mallory in achieving significantly higher state coverage and faster bug-finding speed than the state-of-the-art tool Jepsen. Finally, Mallory discovered 22 previously unknown bugs (10 new vulnerabilities amongs them) which have contributed to new CVEs.

# ACKNOWLEDGMENTS

This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

# REFERENCES

[1] 2023. *Namazu*. https://github.com/osrg/namazu Accessed on 10 April 2023.

[2] Peter Alvaro and Severine Tymon. 2018. Abstracting the geniuses away from failure testing. *Communication ACM* 61, 1 (2018), 54–61. https://doi.org/10.1145/3152483

[3] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy*.

[4] Jinsheng Ba, Marcel Böhme, aZahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful Greybox Fuzzing. In *31st USENIX Security Symposium*.

[5] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful Greybox Fuzzing. In *Proceedings of the 31st USENIX Security Symposium*. USENIX Association, 3255–3272. https://www.usenix.org/conference/usenixsecurity22/presentation/ba

[6] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. 2021. Twins: BFT Systems Made Robust. In *Proceedings of the 2021 Conference on Principles of Distributed Systems (LIPIcs, Vol. 217)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:29. https://doi.org/10.4230/LIPIcs.OPODIS.2021.7

[7] Nathalie Bertrand, Vincent Gramoli, Igor Konnov, Marijana Lazic, Pierre Tholoniat, and Josef Widder. 2022. Holistic Verification of Blockchain Consensus. In *Proceedings of the 36th International Symposium on Distributed Computing (LIPIcs, Vol. 246)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 10:1–10:24. https://doi.org/10.4230/LIPIcs.DISC.2022.10

[8] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D. Ernst. 2020. Visualizing Distributed System Executions. *ACM Transactions on Software Engineering and Methodology* 29, 2 (2020), 9:1–9:38. https://doi.org/10.1145/3375633

[9] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2021. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2021), 79–86. https://doi.org/10.1109/MS.2020.3016773

[10] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1032–1043. https://doi.org/10.1145/2976749.2978428

[11] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *Proceedings of the 29th USENIX Security Symposium*. USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[12] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 159:1–159:30. https://doi.org/10.1145/3276529

[13] Cezara Dragoi, Constantin Enea, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. 2020. Testing consensus implementations using communication closure. *Proceedings of ACM on Programming Languages* 4, OOPSLA (2020), 210:1–210:29. https://doi.org/10.1145/3428278

[14] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2021. FormatFuzzer: Effective Fuzzing of Binary File Formats. *CoRR* abs/2109.11277 (2021). arXiv:2109.11277 https://arxiv.org/abs/2109.11277

[15] Colin J. Fidge. 1988. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66. http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf

[16] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *Proceedings of the 12th European Conference on Computer Systems*. ACM, 328–343. https://doi.org/10.1145/3064176.3064183

[17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. ACM, 1–17. https://doi.org/10.1145/2815400.2815428

[18] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[19] Johansen Hu. 2021. Membership Rollback Issue in Canonical Raft. GitHub issue available at https://github.com/canonical/raft/issues/250.

[20] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *Comput. Surveys* 41, 4 (2009), 21:1–21:54. https://doi.org/10.1145/1592434.1592438

[21] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin Vahdat. 2007. Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 179–188. https://doi.org/10.1145/1250734.1250755

[22] Kyle Kingsbury. 2021. Elle: Finding Isolation Violations in Real-World Databases. In *Proceedings of the 40th ACM Symposium on Principles of Distributed Computing*. ACM, 7. https://doi.org/10.1145/3465084.3467483

[23] Kyle Kingsbury. 2023. *Jepsen*. https://jepsen.io/ Accessed on 10 April 2023.

[24] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Communication ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[25] Leslie Lamport and Stephan Merz. 1994. Specifying and Verifying Fault-Tolerant Systems. In *Proceedings of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (LNCS, Vol. 863)*. Springer, 41–76. https://doi.org/10.1007/3-540-58468-4_159

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

[27] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 399–414. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa

[28] Jeffrey F. Lukman et al. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *Proceedings of the 14th European Conference on Computer Systems*. ACM, 20:1–20:16. https://doi.org/10.1145/3302424.3303986

[29] Rupak Majumdar and Filip Niksic. 2018. Why is Random Testing Effective for Partition Tolerance Bugs? *Proc. ACM Program. Lang.* 2, Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages, Article 46 (2018), 24 pages. https://doi.org/10.1145/3158134

[30] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. 2020. Ankou: guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 1024–1036. https://doi.org/10.1145/3377811.3380421

[31] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Proc. Workshop on Parallel and Distributed Algorithms*. 215–226.

[32] Ruijie Meng, Zhen Dong, Jialin Li, Ivan Beschastnikh, and Abhik Roychoudhury. 2022. Linear-time Temporal Logic guided Greybox Fuzzing. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, 1343–1355. https://doi.org/10.1145/3510003.3510082

[33] Roberto Natella. 2022. StateAFL: Greybox Fuzzing for Stateful Network Servers. *Empirical Software Engineering* 27, 191 (2022). https://doi.org/10.1007/s10664-022-10233-3

[34] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX Association, 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

[35] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of ACM on Programming Languages* 2, OOPSLA (2018), 160:1–160:28. https://doi.org/10.1145/3276530

[36] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 180:1–180:29. https://doi.org/10.1145/3360606

[37] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th annual ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 614–630. https://doi.org/10.1145/2908080.2908118

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *ICST*. 460–465. https://doi.org/10.1109/ICST46399.2020.00062

[39] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th Network and Distributed System Security Symposium*. The Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[40] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 18th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 583–594. https://doi.org/10.1145/3540250.3549139

[41] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. 2018. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 662–677. https://doi.org/10.1145/3192366.3192414

[42] Christopher J.C.H Watkins. 1989. *Learning from delayed rewards*. Ph. D. Dissertation. King's College, Cambridge United Kingdom.

[43] Christopher J.C.H. Watkins and Peter Dayan. 1992. Q-learning. *Machine learning* 8 (1992), 279–292.

[44] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas E. Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs*. ACM, 154–165. https://doi.org/10.1145/2854065.2854081

[45] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 213–228. http://www.usenix.org/events/nsdi09/tech/full_papers/yang/yang.pdf

[46] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 249–265. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan

[47] Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 1141–1156. https://doi.org/10.1145/3373376.3378484

[48] Michał Zalewski. 2023. *american fuzzy lop (2.52b)*. https://lcamtuf.coredump.cx/afl/ Accessed on 10 April 2023.

[49] Jingyu Zhou et al. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 ACM SIGMOD/PODS International Conference on Management of Data*. ACM, 2653–2666. https://doi.org/10.1145/3448016.3457559