

TAMIAS: Feedback-Guided Systematic Concurrency Exploration for Transparent Distributed System Fuzzing

Congyu Liu
Purdue University
West Lafayette, IN, USA

Yaoxu Song
Purdue University
West Lafayette, IN, USA

Pedro Fonseca
Purdue University
West Lafayette, IN, USA

Yongle Zhang
Purdue University
West Lafayette, IN, USA

Abstract

Existing distributed system fuzzers are limited by coarse-grained, nondeterministic event control and weak support for systematic schedule mutation. They rely on timing perturbations (e.g., injecting delays), which can influence execution but do not precisely realize the fine-grained event orderings that trigger many distributed bugs. In addition, they do not integrate systematic schedule exploration with mutation, making exploration less effective in large search spaces.

This paper presents TAMIAS, the first distributed system testing tool to integrate transparent, feedback-guided input exploration with systematic concurrency exploration. TAMIAS is built on two insights: effective distributed-system fuzzing requires feedback determinism rather than full execution determinism, and systematic scheduling algorithms can be integrated with feedback-guided mutation through a replay-explore strategy that allows them to benefit from prior executions while jointly exploring the input and schedule spaces. Guided by these insights, TAMIAS combines lightweight feedback-deterministic transparent event control with feedback-guided input and concurrency exploration, enabling effective distributed system testing.

We evaluate TAMIAS on six widely used distributed systems implemented in C/C++ and Go, without requiring source-code modifications. TAMIAS found 12 new bugs across mature releases. Our ablation study shows that replay-explore mutation improves branch coverage by up to 22% and also improves bug-finding effectiveness. TAMIAS improves branch coverage over the state of the art and achieves feedback determinism for branch coverage on all evaluated target systems.

CCS Concepts: • Software and its engineering → Software testing and debugging; • Security and privacy → Distributed systems security.

Keywords: Distributed systems, Fuzzing



This work is licensed under a Creative Commons Attribution 4.0 International License.

SOSP '26, Prague, Czech Republic

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2585-2/2026/09

<https://doi.org/10.1145/3830418.3843891>

ACM Reference Format:

Congyu Liu, Yaoxu Song, Pedro Fonseca, and Yongle Zhang. 2026. TAMIAS: Feedback-Guided Systematic Concurrency Exploration for Transparent Distributed System Fuzzing. In *ACM SIGOPS 32nd Symposium on Operating Systems Principles (SOSP '26)*, September 29–October 02, 2026, Prague, Czech Republic. ACM, New York, NY, USA, 19 pages. <https://doi.org/10.1145/3830418.3843891>

1 Introduction

Distributed system bugs remain a major threat to modern systems, causing data loss, service downtime, and semantic violations [27, 41, 43]. Such bugs are inherently difficult to detect because they are often triggered by complex combinations of different factors, such as specific user inputs and subtle fine-grained event orderings.

Fuzzing techniques have recently achieved great success in testing system software [9, 13, 19, 24, 25, 52, 66, 73, 80], leading to growing interest in adopting feedback-based fuzzing to test distributed systems [15, 57, 58, 86]. These fuzzers automatically explore the user input space, reducing the developer burden of manual test construction, and use feedback-guided mutation to repeatedly and effectively discover test cases that uncover previously unseen behaviors.

However, while these properties make fuzzing attractive, state-of-the-art distributed system fuzzers suffer from two key design limitations that reduce testing effectiveness [58, 86]. First, they assume that fine-grained, deterministic, and transparent event control is impractical for real-world systems, arguing that it “requires complex OS-scale interposition” [58]. Thus, they provide only nondeterministic, coarse-grained event scheduling: instead of precisely selecting the next enabled event at each program step, such as a specific message delivery, timer event, or thread wakeup, they typically rely on the OS scheduler and coarse perturbation mechanisms, such as injected delays, to indirectly bias event ordering. This control limitation has two consequences. First, it constrains the fuzzer’s ability to trigger bugs that depend on specific fine-grained event interleavings, thereby reducing the scope of bugs that can be found. Second, the lack of control reduces fuzzing stability, making retained test cases less reliable as corpus seeds for subsequent feedback-guided mutation and making it harder to discover deeper execution paths. This point is also emphasized in the official documentation of widely used fuzzers [1, 32, 52, 75].

Second, state-of-the-art approaches do not integrate concurrency exploration algorithms with feedback-guided mutation. Instead, they create event orderings through timing-based perturbations. However, such perturbations provide no clear guarantees about which schedules are explored. Prior work (e.g., PCTCP) [65, 81, 82] on distributed system concurrency testing and our experience have shown that scheduler structure is critical: prior algorithms provide interpretable coverage of the schedule space and stronger coverage guarantees for finding bugs. In turn, stronger coverage guarantees allow testing tools to avoid exploring equivalent states or schedules and to prioritize exploring those that are more likely to uncover bugs according to heuristics based on analyses of real-world bugs, thereby improving effectiveness.

Overcoming these limitations, i.e., coarse-grained, nondeterministic event control and the lack of integration between concurrency exploration algorithms and feedback-guided fuzzing, is challenging. First, achieving deterministic fine-grained event control in real-world systems is difficult because stronger determinism requires controlling increasingly detailed low-level behaviors that the underlying OS itself leaves as nondeterministic, eventually pushing the design toward OS-scale re-implementation. This added complexity in turn reduces compatibility, because the more OS semantics the framework must control, the harder it becomes to support real applications correctly and completely. As a result, such deterministic fine-grained control is rare in prior distributed-system testing work. To our knowledge, Modist [81], a distributed system model checker, is the only exception, although it does not clearly evaluate or quantify the achieved determinism and does not use a feedback-guided mutation-based approach. Second, although concurrency exploration algorithms provide useful exploration properties, they do not naturally fit the feedback-guided mutation workflow of fuzzing. For example, concurrency exploration algorithms such as random walk and PCTCP generate schedules from scratch for each execution, making it unclear how to derive new schedules from prior feedback-selected runs while preserving most of their exploration properties.

This work addresses the challenge of deterministic fine-grained event control and the challenge of integrating mutation with exploration algorithms through two key insights. First, full execution determinism is stronger than effective fuzzing exploration actually requires. Instead, it is sufficient to provide fine-grained event control with *feedback determinism*, i.e., the execution of the same test case produces the same feedback during fuzzing. This relaxed determinism helps fuzzing effectiveness by making the corpus a stable basis for exploration: each retained test reliably reproduces the feedback for which it was retained, allowing subsequent mutations to build on that discovery and explore deeper behaviors. Feedback determinism is sufficient for fuzzing effectiveness because feedback defines the behavior abstraction over which fuzzing explores; nondeterminism invisible

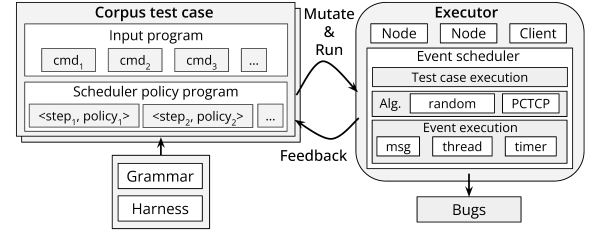


Figure 1. TAMIAS overview.

to this abstraction does not affect the fuzzer’s exploration decisions, while controlling it would increase the complexity of event control. Second, systematic scheduling algorithms need *not* be limited to generating executions from scratch. Instead, mutation can be integrated with scheduling algorithms by operating at different levels, through a principled *replay-explore* strategy that reuses prefixes from interesting, previously discovered tests.

Guided by these insights, we build TAMIAS (Figure 1), the first transparent distributed-system fuzzer that integrates feedback-guided input generation with systematic concurrency exploration. Here, *systematic* means that TAMIAS explicitly controls the execution order of enabled events and selects each event using a feedback-guided exploration algorithm, rather than merely biasing the OS scheduler with sleeps or randomly mutating delays. *Transparent* means that TAMIAS tests unmodified distributed systems.

TAMIAS targets branch coverage determinism since it uses branch coverage as feedback. We show that this fuzzing-oriented determinism goal enables a lightweight transparent fine-grained event-control design. Specifically, TAMIAS combines a transparent network proxy for message interposition, sched_ext with eBPF for thread-scheduling control, and lightweight system-call interception for timers via timeout rewriting. Together with a few additional nondeterminism-mitigation techniques, these mechanisms allow TAMIAS to achieve feedback determinism for branch coverage while preserving compatibility with real-world systems.

TAMIAS’s replay-explore strategy allows it to jointly mutate inputs and event schedules while incorporating existing scheduling algorithms for concurrency exploration. Instead of perturbing individual events with ad hoc delays, TAMIAS first *replays* the prefix of a corpus test case and then *explores* a different suffix by dynamically reconfiguring the scheduler policy through changes to the inputs that guide scheduler decisions, such as the scheduling algorithm’s pseudo-random seed, the set of enabled user inputs, and the virtual clock speed. This lets the fuzzer derive structured schedule variants from prior interesting executions while preserving compatibility with existing concurrency exploration algorithms. It also makes the algorithms naturally compatible with user input mutation: newly inserted user inputs can be enabled only after a chosen scheduling step, allowing earlier execution prefixes to be replayed and reused. Concretely, TAMIAS

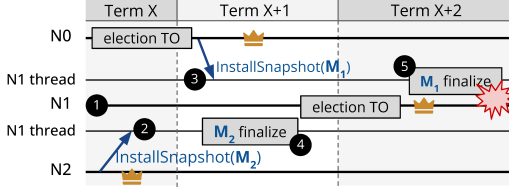


Figure 2. Nuraft-1: A crash bug found in Nuraft [3]. Some events (e.g., voting messages) are omitted for simplicity.

represents a test case as two programs: an input program consisting of external input commands, and a scheduler-control policy program consisting of commands $\langle s, p \rangle$ that apply policy change p at scheduling step s . Mutation operates over these two programs, enabling replay-explore-style joint mutation of inputs and schedules.

We evaluated TAMIAS on six popular distributed systems implemented in C/C++ and Go: Redis, ETCD, MySQL, MongoDB, Nuraft, and Braft, without code modifications. Across mature releases of these systems, TAMIAS found 12 new bugs with serious consequences, ranging from crashes and memory errors to semantic violations. TAMIAS outperforms the state-of-the-art distributed-system fuzzer DistFuzz on the four shared targets, increasing branch coverage by up to 47% and demonstrating the value of fine-grained event control and more systematic schedule exploration. We also conducted an ablation study of our proposed test-case mutation approach, which improves final branch coverage on the majority of targets under both scheduling algorithms by up to 22%, while also increasing bug-finding effectiveness. Our ablation results further suggest that TAMIAS is especially helpful when the input space is large, i.e., when the target systems support a large number of different types of user requests. TAMIAS achieves branch coverage feedback determinism in our evaluation. The TAMIAS artifact is open source [48].

2 Background and Motivation

2.1 Distributed System Fuzzing

Although existing distributed-system fuzzers [57, 58, 86] have shown promise in exploring the input space through feedback-guided mutation, they lack fine-grained control over nondeterministic events and use timing-based event perturbation to generate event schedules.

For instance, Mallory [58], an extension of Jepsen that uses feedback to guide fault injections, does not explicitly control fine-grained internal events such as individual message deliveries, timer firings, or thread wakeups. DistFuzz [86] uses message sequence feedback to mutate client and message events, faults, and their relative timing. However, DistFuzz influences internal event ordering indirectly through timing-based perturbations; TAMIAS instead integrates feedback-guided mutation with a concurrency-exploration algorithm that explicitly selects the next enabled message-delivery, timer, or thread-scheduling event.

This design, based on coarse-grained event control and non-systematic mutation, is insufficient for effective testing. Distributed-system behavior depends jointly on user input and fine-grained event schedules, requiring effective exploration of the combined test space. The ability to deterministically enforce fine-grained schedules is crucial not only for uncovering the specific schedules that trigger subtle bugs, but also for progressively and consistently exploring new behaviors that increase system coverage. To further illustrate this point, we next discuss a case study from a real-world distributed system.

Nuraft bug case study. The Nuraft-1 bug (shown in Figure 2) was found by TAMIAS in Nuraft [3], a Raft consensus implementation used by eBay and several open-source systems, such as ClickHouse. The bug is triggered when node N1 receives two `InstallSnapshot` requests across a leader transition: one finishes before N1 becomes the leader, while another is processed after N1 becomes the leader.

To trigger this bug, user inputs must first grow the Raft log sufficiently and cause N1 to fall behind the leader (1). In term X , node N1 receives an `InstallSnapshot` request, M_2 (2). Before node N1 finishes processing M_2 , node N0 times out, wins a new election, and becomes the leader for term $X + 1$. As the new leader, node N0 sends another `InstallSnapshot` request, M_1 , to node N1 (3). Node N1 then completes processing of snapshot M_2 (4). Afterward, node N1 itself times out, wins the next election, and becomes the leader in term $X + 2$. At this point, as the processing of the `InstallSnapshot` request M_1 is about to finish, the code observes that N1 is now a leader, violating the internal assertion that a leader must not be installing a snapshot (5).

In summary, this bug requires very specific user inputs and a fine-grained event ordering within an execution involving tens to hundreds of messages and thread-scheduling decisions, making it a needle-in-a-haystack search. Even a principled scheduling algorithm such as PCTCP may struggle to discover such an ordering efficiently, as its probability guarantee weakens with increasing bug depth and execution length [65]. In §5.2, we show how mutation helps expose this bug by letting PCTCP build on past successful executions, rather than restarting from scratch, and gradually reach deeper bug-triggering states.

2.2 Our Insights

As discussed, enforcing deterministic fine-grained schedules and integrating feedback-based mutation with systematic schedule exploration are two key challenges in effective testing. TAMIAS addresses these challenges with two insights.

Insight 1: Fine-grained scheduling using feedback determinism is sufficient and practical. Our goal for determinism is to support effective feedback-guided exploration, rather than to reproduce every execution detail as required by deterministic debugging [62]. For fuzzing, the key role

of determinism is to provide a stable corpus [1, 32, 52, 75]: a test is retained because it reaches new feedback, and subsequent mutations use that test as a basis for exploring further behaviors. If rerunning a retained test does not reproduce that feedback, the corpus no longer provides a reliable starting point for exploration. For example, if a retained test no longer reproduces the feedback reached after partitioning the leader, appending a write may no longer exercise the intended failover behavior. Thus, greybox fuzzing requires retained feedback to be reproducible. Since feedback defines the behavior abstraction over which the fuzzer explores, making this feedback deterministic already stabilizes the fuzzer's exploration space; controlling execution determinism invisible to feedback provides no additional stability for this purpose.

This relaxed determinism requirement is important in practice. Pursuing full execution determinism often requires broad interception and emulation of OS semantics, which increases overhead and enlarges the compatibility surface for real-world applications. For large, feature-rich distributed systems, this can make transparent support difficult to provide. By requiring only feedback determinism, we show that a distributed system fuzzer can use a simpler event-control design. For example, TAMIAS uses a lightweight transparent proxy for network interception rather than emulating the full network stack, avoiding both the large network system-call surface and the complexity of full protocol emulation. Another example is the complex `procf's` interface, which TAMIAS currently does not emulate because it does not usually cause instability in branch-coverage feedback.

Insight 2: Integrating scheduling algorithms with mutation through a replay-explore strategy. Our second insight is that, instead of generating each event schedule from scratch, the fuzzer can guide the scheduling algorithm to produce a mutated schedule using a principled *replay-explore* strategy. Given an event schedule from the corpus, the fuzzer first lets the scheduling algorithm replay an execution prefix using the original scheduling-policy configuration, such as the same PRNG seed and enabled inputs. It then reconfigures the scheduler during execution, for example, by changing the PRNG seed or enabling new inputs, so that the remaining execution diverges into a different schedule and explores different behaviors. This lets the fuzzer reuse schedules that have already exposed interesting behavior, such as previously uncovered branches, and continue exploration from there toward deeper behaviors.

3 TAMIAS Design

TAMIAS targets unmodified, multithreaded distributed systems that interact through message passing. TAMIAS controls and observes the system under test, allowing it to intercept various events and manipulate their ordering. TAMIAS can also mutate system inputs, e.g., client requests, and observe

Algorithm 1 TAMIAS fuzzing loop.

Require: Grammar G , scheduling algorithm A

```

1:  $corpus \leftarrow \{\langle \text{CREATEINPUT}(G), \text{RANDSCHEDULE}(A) \rangle\}$ 
2: repeat
3:    $\langle I, \sigma \rangle \leftarrow \text{SELECTSEED}(corpus)$   $\triangleright$  pick seed test case
4:    $s \leftarrow \text{CHOOSESTEP}(\sigma)$   $\triangleright$  pick divergence point in schedule
5:    $I' \leftarrow \text{MUTATEINPUT}(I, G, s)$   $\triangleright$  mutate input after  $s$ 
   REPLAY
6:    $P \leftarrow \text{GETPOLICIES}(\sigma_{[0,s]})$   $\triangleright$  reuse the same policies
7:    $\text{RUN}(A, I, P, 0, s)$   $\triangleright$  replay prefix  $[0, s]$ 
   EXPLORE
8:    $(s, \tilde{p}_a) \leftarrow \text{NEWSCHPOLICY}(A)$   $\triangleright$  new scheduling policy
9:    $(s, \tilde{p}_c) \leftarrow \text{NEWLOCKPOLICY}()$   $\triangleright$  new clock policy
10:   $(s, \tilde{p}_i) \leftarrow \text{NEWINPUTPOLICY}(I')$   $\triangleright$  new input policy
11:   $\tilde{P} \leftarrow \{(s, \tilde{p}_a), (s, \tilde{p}_c), (s, \tilde{p}_i)\}$   $\triangleright$  update policies
12:   $\tilde{\sigma}_{[s,\infty)} \leftarrow \text{RUN}(A, I', \tilde{P}, s, \infty)$   $\triangleright$  explore after  $s$  with policy  $\tilde{P}$ 
   COLLECT FEEDBACK
13:  if  $\text{HASNEWFEEDBACK}()$  then
14:     $corpus \leftarrow corpus \cup \{\langle I', \sigma_{[0,s]} \parallel \tilde{\sigma}_{[s,\infty)} \rangle\}$ 
15:  end if
16: until time budget exhausted

```

the resulting behavior, including failure symptoms and feedback for fuzzing.

During test execution, TAMIAS controls both *internal events* generated by the target system and *external events* that are introduced from outside the target system. Internal events are TCP message deliveries, timer firings, and thread scheduling events. §3.6 describes how TAMIAS interposes on each event source. External events fall into two groups: The first group consists of client-request and partition-fault events, which the fuzzer generates and mutates as test inputs. Partition faults are injected between nodes in the target system and affect their internal messages. Other external events, namely start process and signal process events, are used for test orchestration. For example, a start process event can launch a server process on a designated container host.

3.1 TAMIAS Overview

Figure 1 provides an overview of the TAMIAS testing environment. Unlike conventional single-node fuzzing, where a test case typically consists only of program inputs, distributed-system fuzzing must also control how nondeterministic events are scheduled. A TAMIAS test case therefore consists of two parts: an input program I and a schedule-control program P . The input program I is a sequence of external commands, e.g., client requests or network partition faults. The schedule-control program P is a sequence of pairs $\langle s, p \rangle$, where s is a scheduling step and p is a control action to apply at that step, such as reseeding the scheduler or enabling newly added inputs. When a test case is executed, the scheduling algorithm A and virtual clock use these control actions together with enabled events to generate a realized execution schedule σ , consisting of message deliveries, timer expirations, thread scheduling decisions, and external events. The

Algorithm 2 $\text{RUN}(A, I, P, \text{start}, \text{end})$.

Require: Algorithm A , input I , policies P , step range $[\text{start}, \text{end}]$

```

1: for  $s \leftarrow \text{start}$  to  $\text{end} - 1$  do
2:    $\text{APPLYPOLICY}(A, I, \text{clk}, P[s])$   $\triangleright$  apply policies at  $s$  (§3.4)
3:    $e \leftarrow A.\text{NEXT}()$   $\triangleright$   $A$  picks event
4:    $\text{new} \leftarrow \text{RUNANDINTERCEPT}(e)$   $\triangleright$  execute  $e$ ; intercept new
    events; wait for quiescence (§3.6)
5:    $A.\text{ADDEVENTS}(\text{new})$   $\triangleright$  add new events to  $A$ 
6:   if  $\text{FINISHED}()$  then break  $\triangleright$  requests complete or timeout
7: end for
8: return observed schedule  $\sigma'$ 
```

schedule-control program is a unified abstraction that describes how the scheduler’s components guide an execution: at selected scheduling steps, it reconfigures the scheduling algorithm or virtual clock, while each scheduling algorithm defines how algorithm-specific policies reconfigure its state.

TAMIAS supports two kinds of input commands: client requests and fault commands, each mutated with type-specific mutators. A client request consists of a destination node index and a request string. The destination is generated either randomly or by reusing node indices from existing requests, while the request string is generated or mutated from the user-provided grammar using grammar-based mutation [5]. Note that the eventual request destination depends on the client harness and the system’s routing logic. For SQL workloads, TAMIAS also integrates SQLSmith [67]. For fault commands, the current prototype supports commands that create and heal network partitions over selected host pairs.

At runtime, the scheduler translates each input command into an external event, such as a client-request event or a network-partition event. To execute a client-request event, the scheduler sends a control message over TCP instructing the client harness to issue the corresponding request. The network partition fault event implementation is described in §3.6.3. Each external event is assigned a unique ID, which the schedule-control program can later reference to determine when that event becomes enabled.

3.2 TAMIAS Fuzzing Loop

Algorithm 1 shows the high-level fuzzing loop of TAMIAS. It follows the standard greybox fuzzing pattern of seed selection, mutation, execution, and feedback. As shown in line 1, the algorithm starts with an initial corpus containing a single test case, where the input is generated by `CreateInput` using the system’s input grammar and the schedule is generated by `RandSchedule` with the selected scheduling algorithm. In each iteration, TAMIAS selects a seed test case from the corpus (line 3) and then chooses a step s in the schedule σ as the divergence point (line 4).

For clarity, Algorithm 1 uses (I, σ) as a high-level representation of a test, pairing its input with its previously observed scheduling trace. This abstraction makes replay and exploration easier to describe. In the implementation,

```

1 type SchedulingAlgorithm interface {
2   AddEvents(events []Event) // add events to the algorithm
3   Next() Event              // get the next event
4   Reconfig(policy any)      // apply a new policy
5 }
```

Figure 3. The scheduling algorithm interface.

TAMIAS instead represents each test case as (I, P) , where P is a step-indexed program of policy updates. We next describe how these policies realize replay and exploration.

TAMIAS **replays** the selected test case through step s by running its input I and, at each scheduler-policy-update step recorded in P , reconfiguring the scheduler with the original policies: algorithm-specific policies (e.g., a PRNG seed), input policies (e.g., which client requests are enabled), and virtual-clock policies (i.e., when timers are selected) (lines 6–7). After replaying the prefix, TAMIAS reconfigures the scheduler with fresh policies and runs the system with the mutated input—specifically, the new commands in the mutated input are added beginning at step s , allowing the execution to diverge and **explore** new behavior (lines 8–12). Finally, the fuzzer compares the new execution’s branch coverage with the global coverage bitmap and adds the mutated test case to the corpus if it triggers a previously unseen branch (lines 13–14).

3.3 Replay-Explore Mutation Strategy

The replay-explore mechanism described above addresses a fundamental mismatch between feedback-based test case mutation in greybox fuzzing and schedule generation in distributed system concurrency exploration algorithms: Traditional greybox fuzzing reuses interesting seeds by mutating them, but concurrency exploration algorithms such as random walk and PCTCP are designed to generate complete schedules from scratch. As a result, an interesting distributed execution from a previous test is difficult to reuse directly: re-running from the initial state wastes effort rediscovering the same prefix, while mutating the input may perturb execution so early that the useful state is never reconstructed.

TAMIAS addresses this mismatch with replay-explore mutation. Replay provides *local search*: the shared prefix constrains the new execution to re-enter the same execution neighborhood as the original corpus seed, rather than searching again from the initial state. The suffix is then explored systematically by the scheduling algorithms rather than by ad hoc timing perturbation. After step s , TAMIAS reconfigures the scheduler, allowing the scheduling algorithm to explore a new suffix from the reconstructed state according to its exploration policy.

To make input mutation compatible with replay, TAMIAS delays newly added input commands until step s . The shared part of the input reconstructs the replayed prefix, while the new commands are enabled only when exploration begins. In this way, both schedule and input mutation preserve the

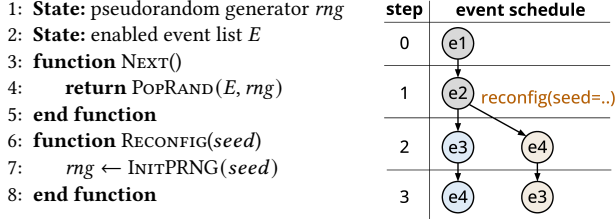


Figure 4. Random-walk reconfigurations.

replayed prefix while introducing new choices only in the explored suffix. Replay-explore therefore combines the two ingredients that distributed-system fuzzing needs: local search around previously interesting executions and systematic exploration of new schedule suffixes from those reconstructed states (the preserved properties are described in §3.5).

3.4 Scheduling Algorithm Interface

TAMIAS is designed to support different scheduling algorithms. To achieve this, TAMIAS defines a scheduling algorithm interface as shown in Figure 3. Any scheduling algorithm that implements this interface can be plugged into TAMIAS. The interface includes methods for adding events, picking the next event, and reconfiguring the scheduler with different policy parameters, such as a new seed. `AddEvents` is invoked when new events arrive and passes them to the scheduling algorithm for scheduling. `Next` returns the next event to be scheduled. This allows TAMIAS to support different scheduling algorithms, such as random walk and PCTCP.

The `Run` procedure in Algorithm 2 connects the main fuzzing loop to the scheduling algorithm. It takes a scheduling algorithm *A*, an input *I*, a scheduler-control policy program *P*, and a step range $[start, end)$ as arguments. At each step *s*, it uses `ApplyPolicy` to apply every command in $P[s]$ before *A.Next* selects an event. It then executes the event, intercepts any newly enabled events, and adds them back to *A*, allowing schedule generation to adapt as execution proceeds. Finally, `Run` returns the observed schedule σ' for further exploration.

`ApplyPolicy` loops over $P[s]$ and dispatches each command by type (see Algorithm 4 in §A.3). Algorithm policies call *A.Reconfig*, input policies add referenced events through *A.AddEvents*, and virtual-time policies update the timer-selection probability. §3.5 further describes algorithm-specific policies.

3.5 Scheduling Algorithm Policy Mutation

`Reconfig` is the core interface for applying algorithm-specific policy changes at the replay-explore boundary. It should preserve the scheduler state fixed by the replayed prefix and modify only the state governing future calls to `Next`. Specifically, `Reconfig` resamples future random choices so that the resulting schedule suffix can diverge from the original, while preserving the underlying scheduler's distribution over possible suffixes from the replayed state under the new policy

Algorithm 3 PCTCP reconfiguration.

Require: New seed *seed*, event count *n*, bug depth *d*, step *s*

```

1: rng  $\leftarrow$  INITPRNG(seed) ▷ new seed
2:  $C_p \leftarrow \{c \in C \mid c < s\}$  ▷ past change points
3:  $C_f \leftarrow \text{SAMPLEDISTINCT}([s, n], d - 1 - |C_p|)$  ▷ resample
4:  $C \leftarrow C_p \cup C_f$ 

```

parameters. We formalize this guarantee below as the suffix-exploration property, which ensures that replay does not weaken the scheduler's exploration from the replayed state.

Figure 4 shows an example of `Reconfig` for the random-walk scheduler, which reconfigures the scheduler's pseudorandom number generator (PRNG) seed. As shown on the left, the random walk scheduler uses a PRNG to randomly select the next enabled event. The scheduler's `Reconfig` implementation creates a new PRNG with the given seed. Hence, as shown on the right, calling `Reconfig` before step 2 replaces the original PRNG with a new one, which can cause the event schedule to diverge from the original after step 2.

TAMIAS also supports PCTCP [65] (Probabilistic Concurrency Testing with Chain Partitioning), a randomized distributed event scheduler with probabilistic guarantees, and provides the corresponding `Reconfig` implementation. PCTCP is a generalization of PCT [11] for distributed systems: it organizes events into chains of causally related events (e.g., a message and the events it triggers), and assigns each chain a random priority. At each step, PCTCP picks the first unexecuted event from the highest-priority chain. The key mechanism is priority change points: PCTCP randomly selects $d - 1$ scheduling steps at which it decreases the priority of the currently executing chain, where *d* is a user-provided bug-depth parameter. This guarantees that any bug requiring at most *d* ordering decisions is found with probability $p \geq 1/w^2 n^{(d-1)}$ per execution, where *n* and *w* denote the event count and scheduling-poset width. PCTCP initially generates these priority change points by randomly sampling scheduling steps from the range determined by the total number of events.

As shown in Algorithm 3, PCTCP preserves the priority changes already applied before step *s*. Using a fresh seed, it samples the remaining distinct change points from $[s, n)$ based on the given *n* and *d*. TAMIAS may change *n* for events introduced by input mutation or *d* for deeper bugs.

Suffix-exploration property. Replaying the prefix of a useful corpus seed can make TAMIAS more likely to revisit a state needed to trigger a bug or other interesting behavior, instead of rediscovering that state from scratch. This advantage is retained only if replay-explore does not weaken the underlying scheduler's suffix exploration from that state.

To see why, let S_s denote the system state after executing $\sigma_{[0,s)}$, together with the scheduler state preserved when the new policies are applied at step *s*. Suppose a bug requires first reaching S_s and then generating a bug-triggering suffix. The probability of finding the bug through S_s is the probability

of reaching S_s multiplied by the conditional probability of generating such a suffix from S_s . We show that the Reconfig implementations for random walk and PCTCP preserve the second factor: once S_s is reached, replay–explore gives every possible suffix the same probability as A alone.

We formalize the suffix-exploration property as follows. Fix a replayed state S_s , mutated inputs I' , future clock and input policies $\tilde{P}_c$ and $\tilde{P}_i$, and the final non-random policies θ_A of A (e.g., parameters n and d for PCTCP). Let $C_s = (S_s, I', \tilde{P}_c, \tilde{P}_i, \theta_A)$ denote this suffix-generation context. Let $\text{Pr}_{\text{RE}(A)}$ and Pr_A denote probabilities when the suffix is generated from C_s by replay–explore instantiated with A and by A alone, respectively. Then, for every possible schedule suffix σ' ,

$$\text{Pr}_{\text{RE}(A)}[\tilde{\sigma}_{[s,\infty)} = \sigma' \mid C_s] = \text{Pr}_A[\tilde{\sigma}_{[s,\infty)} = \sigma' \mid C_s].$$

We next show that the random-walk and PCTCP reconfigurations satisfy this property.

Proof sketch. For random walk, Reconfig installs a fresh PRNG whose future random choices follow the algorithm’s original distribution. For PCTCP, it preserves the existing chain state and past priority change points, and resamples all unused random choices, such as distinct future change points from $[s, n)$ under the final n and d . Thus, from any current state, replay–explore and A alone induce the same distribution over the event selected by A . Next and the resulting scheduler state. Executing that event and adding newly enabled events then induces the same distribution over the next system and scheduler state. Induction over subsequent scheduling steps gives the suffix equality. Note that this property is not guaranteed by arbitrary mutations of a scheduler’s future random choices. For example, mutating only a few entries in a recorded random-choice sequence, as in Fest [44], leaves the remaining choices fixed and therefore does not preserve the scheduler’s suffix distribution.

Relation to replay determinism. Replay determinism and suffix exploration address two separate problems. Replay determinism determines the possible states reached after the prefix: a fully deterministic replay would always reach the same state, while feedback-deterministic replay may reach different states that produce the same feedback (e.g., branch coverage). The suffix-exploration property is conditional on the state reached, so prefix nondeterminism may change where exploration starts but not the equality between the suffix distributions induced by replay–explore and by A alone from that state.

3.6 Feedback-Deterministic Event Interception

TAMIAS’s replay–explore strategy requires the scheduler to transparently intercept and control messages, timers, and thread interleavings on unmodified systems. For each event type, TAMIAS uses a mechanism chosen to minimize the interposition surface while providing feedback determinism.

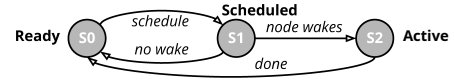


Figure 5. TAMIAS scheduler state transitions.

Specifically, TAMIAS (1) intercepts messages using an external transparent TCP proxy built on iptables, (2) controls thread interleavings at the kernel scheduler level using `sched_ext` (a Linux kernel scheduler extension) via eBPF, and (3) controls timers with a time-dilation-style virtual clock [34] to avoid timer anomalies. This section explains how TAMIAS transparently and deterministically controls events at scale in unmodified systems as large as MySQL (5.9M LOC) and MongoDB (7.3M LOC).

3.6.1 Overview. TAMIAS runs the scheduler and target processes inside Nyx-Net [72] virtual machines. It uses Nyx-Net’s snapshot-reload support to restore an initialized cluster before each test, avoiding repeated cluster initialization. Each host is instantiated as a container with an IP address from a user-provided container image and may run multiple target processes. The scheduler intercepts events using different interception mechanisms and monitors target processes via eBPF to determine when to schedule the next event.

3.6.2 Scheduling Model. Scheduling proceeds in a sequence of *scheduling steps*. TAMIAS uses a run-to-completion model: at each step, the scheduler selects one enabled pending event and lets it execute until its processing completes or blocks. During execution, the scheduler intercepts newly generated events and adds them to the pending set. To prevent a long-running event, such as a spinning thread, from monopolizing execution, TAMIAS may preempt it. Figure 5 shows the scheduler state machine, which transitions among **S0** (Ready), **S1** (Scheduled), and **S2** (Active). Each scheduling step begins in **S0**.

Detecting activity. When an event is scheduled, the scheduler enters **S1** and waits for the system to become active, i.e., for any thread to become runnable. When a thread becomes runnable, the scheduler transitions to **S2**.

No wake detection. In some cases, delivering a message does not immediately wake any thread. For example, the message processing thread may be blocked on a lock rather than polling for new messages. After transitioning to **S1**, TAMIAS therefore needs to infer whether any thread will wake to process the message, since it does not directly intercept and emulate the network system call interface. Unfortunately, the kernel does not provide a direct hook that explicitly indicates “this receive will not wake a thread.”

We leverage the fact that the kernel makes scheduling decisions (i.e., whether threads will wake up) during the receive path, before the corresponding `softirq` handler returns. Thus, when scheduling a message event, TAMIAS uses the return of the associated `softirq` as a barrier: if no wakeup occurs before this point, TAMIAS concludes that no thread

will wake up to process the message and transitions from S_1 to S_0 .

Quiescence detection. After a thread begins running (at S_2), TAMIAS employs quiescence detection to determine when to schedule the next event, i.e., when to transition to S_0 . Specifically, it considers the system to be in a quiescent state when there are (1) no runnable threads and (2) no in-flight messages between the target system and TAMIAS's network proxy.

Implementation details. TAMIAS implements a monitor component that tracks the system status. It uses eBPF [22] to collect execution information from the target system. TAMIAS uses kernel scheduling tracepoints to collect process activity information. Specifically, it traces when a process's runnable state changes (e.g., `runnable()` in `sched_ext`). TAMIAS uses kernel network tracepoints (e.g., `tcp_sendmsg`) to monitor network activity, such as the number of bytes sent or received and connection establishment or termination. Compared with prior work, this approach avoids expensive waiting [43] and does not require target-side network emulation [56].

3.6.3 Network Interposition. TAMIAS intercepts and forwards network messages through an L4 transparent TCP proxy. The proxy preserves original source addresses, which is necessary because some distributed systems require messages from other nodes to match configured peer addresses. It also preserves FIFO delivery on each TCP connection, avoiding violations of TCP assumptions and false bugs.

Benefit. This design has two advantages. First, it produces fewer, coarser events (logical messages rather than individual socket operations), reducing the scheduling search space. Second, not emulating the Linux socket API yields a compact (420 LOC) network component suitable for large systems.

Remote traffic. To intercept remote traffic destined for other nodes, TAMIAS changes the default gateway for such traffic so that it passes through the proxy container. Inside the proxy container, TAMIAS uses the iptables `TProxy` target to capture the traffic. When the proxy intercepts a connection whose source and destination addresses belong to nodes or clients, it binds to the connection's source address and establishes a new connection to the destination address.

Local traffic. Local traffic bypasses the default gateway via the local route table [10]. TAMIAS uses destination NAT (DNAT) to redirect local messages to the proxy, then source NAT (SNAT) to hide the proxy address and maintain the illusion of direct node-to-node communication.

Network partition fault. TAMIAS supports creating and healing network partitions between two nodes. To create a network partition between two nodes, TAMIAS (1) configures the firewall to prevent new connections from being established and (2) instructs the scheduler not to schedule messages between the isolated nodes.

3.6.4 Timer Interposition. Distributed systems rely heavily on timeouts for failure detection, leader election, and protocol progression. TAMIAS controls timers by virtualizing the system's entire perception of time, an approach inspired by time dilation in system emulation [34]: the runtime controls how virtual time advances, and timeout behavior follows from that virtual clock. Note that TAMIAS does *not* treat timers as freely permutable events. Instead, it controls when virtual time advances relative to other events; among timers themselves, expiration order continues to follow deadline order. Currently, TAMIAS does not support clock skew. Supporting this would require managing virtual time on a per-node basis, which is left for future work.

Benefit. This choice preserves timer semantics and avoids invalid executions caused by arbitrarily reordering timeouts. For example, protocols based on leases or election timeouts rely on deadline relationships, not just on the existence of timers. If the runtime could simply fire any pending timeout at will, it could violate those relationships. By advancing virtual time and firing the earliest due timeout, TAMIAS explores how timeouts interleave with message and thread events while keeping timer ordering consistent with the target's virtual-time semantics.

Timer emulation. Distributed systems often rely on timeouts to implement timers. A common pattern is to use polling system calls (e.g., `epoll_wait`, `poll`) with a timeout or sleep, so that a thread wakes up periodically even if no new events arrive. TAMIAS's interception runtime rewrites these timeout arguments to "infinite", records the intended timeout with the scheduler, and then issues the system call. When the scheduler decides to fire the timer, it advances the virtual clock and signals the corresponding thread with a special real-time signal. Upon handling the signal, the interception runtime rewrites the system call's return value as if the thread had been woken by the timeout.

Time emulation. TAMIAS intercepts time-related interfaces, including calls such as `clock_gettime`, as well as timestamp-counter instructions such as `RDTSCP`, and rewrites their results to reflect virtual time. At a high level, virtual time is computed as the start time plus scheduler-advanced time. Crucially, this value is computed efficiently in the target process without contacting the scheduler: the process reads scheduler-controlled time state from shared memory.

Implementation details. Timer control is implemented using an in-process runtime that intercepts related system calls. Specifically, TAMIAS uses `LD_PRELOAD` to transparently inject the runtime into the target system and intercept libc-based system calls. For system calls that originate outside libc, TAMIAS installs a seccomp [76] filter and registers a `SIGSYS` handler to process the selectively intercepted system calls. Since vDSO-based system calls [46] are not trapped by the kernel and thus escape the seccomp filter, TAMIAS dynamically patches the vDSO system calls to jump into the

runtime. Together, these three mechanisms ensure complete interception of all necessary system calls.

3.6.5 Thread Scheduling. TAMIAS uses `sched_ext` to control thread scheduling without kernel modifications. The scheduler has two components. The in-kernel `sched_ext` program acts as a lightweight data plane: it intercepts runnable threads and enforces scheduling decisions from user space. The user-space scheduler serves as the control plane and decides which thread to run next.

Benefit. This approach is more general than intercepting individual thread synchronization APIs (e.g., `pthread` or system calls such as `futex`): it captures all sources of thread runnability through a single kernel hook, without needing to enumerate and wrap each synchronization primitive. It intercepts all events that make a thread runnable, not just explicit synchronization events.

Implementation details. At initialization, the in-process interception runtime invokes `sched_setscheduler` to place target threads under TAMIAS’s `sched_ext` scheduling policy. When a thread becomes runnable, `sched_ext`’s enqueue hook fires and notifies the user-space scheduler of the runnable thread. When the user-space scheduler selects a thread to run, it signals `sched_ext`, which dispatches the chosen thread to a CPU (via `dispatch`). To enforce run-to-completion, TAMIAS assigns an effectively infinite timeslice to avoid unnecessary preemptions. Running one thread at a time can still expose data-race bugs through different serialized thread interleavings. TAMIAS does not explore instruction-level thread interleavings due to the larger search space. We leave finer-grained thread interleaving exploration to future work.

3.6.6 Mitigating Non-Determinism. TAMIAS must mitigate additional sources of nondeterminism to achieve feedback determinism. While the prior sections describe how TAMIAS controls nondeterminism through event interception and scheduling, this section details other sources of nondeterminism we identified and the corresponding mitigations.

Network non-determinism. Network nondeterminism can still arise between the TAMIAS proxy and the target system. TCP may coalesce or split application messages across packets, so TAMIAS merges all messages on the same connection within a scheduling step into one logical message event. Nagle’s algorithm can also delay small packets nondeterministically, so TAMIAS disables it by intercepting socket calls and setting `TCP_NODELAY` with `setsockopt`. Finally, a thread may send messages over multiple connections during a scheduling step. Although their kernel send order is deterministic because TAMIAS runs at most one thread at a time, the messages may reach the proxy in a different order. TAMIAS therefore uses the eBPF send trace to reconstruct the kernel send order and enforce it at the proxy.

Thread transient sleep. During execution, a thread may nondeterministically block in the kernel while waiting for

a resource; we refer to such short, unintended blocking as *transient sleep*. For example, a thread may briefly block in `fsync` while waiting for page writeback. This issue arises because TAMIAS uses `sched_ext` to observe and control thread execution even when it enters the kernel. Transient sleep is problematic because it is visible to TAMIAS but can be mistaken for an intended scheduler-controlled blocking point. From the scheduler’s perspective, the thread appears to sleep while waiting for the next modeled event, such as a message delivery or timer expiration. However, the sleep is actually caused by a transient kernel-internal resource wait, and the thread may later wake up without any corresponding event in TAMIAS’s execution model. This violates the assumption that thread blocking and wakeup are driven by explicit modeled events, and it further introduces nondeterminism across runs. To mitigate this issue, TAMIAS maintains a list of kernel functions that commonly block on transient resource waits, such as page-fault handling and disk writeback. If a sleep occurs while executing one of these functions, TAMIAS classifies it as transient and does not report it to the scheduler.

Entropy sources. To eliminate randomness-induced nondeterminism, TAMIAS controls both OS- and CPU-level entropy sources. For OS-provided randomness, TAMIAS returns deterministic values for `/dev/random`, `/dev/urandom`, and related interfaces such as `getrandom`. For CPU entropy, TAMIAS applies a trap-and-emulate strategy inspired by `rr` [62]: it intercepts `CPUID` and masks out the `RDSEED` and `RDRAND` feature bits so the target system observes these instructions as unsupported and follows its fallback path. Some applications may also use process IDs as a source of entropy (e.g., Redis). Hence, TAMIAS assigns deterministic PIDs by launching target processes in a separate PID namespace.

3.6.7 Discussion: Comparison with Modist. Modist [81] achieves the same high-level goal—transparent control over messages, timers, and threads—by uniformly wrapping OS APIs (socket APIs for messages, thread APIs for threads, and time APIs for timers). Compared to Modist, TAMIAS’s design differs mainly in two ways: (1) the TCP proxy produces fewer events and avoids the socket API surface; and (2) `sched_ext` provides generic kernel-level thread control without enumerating synchronization APIs.

3.6.8 Discussion: Feedback Determinism. We define *feedback determinism* as the property that repeated executions of the same test case produce the same feedback. In TAMIAS, the feedback is branch coverage. To achieve this property across our six target systems, TAMIAS controls message delivery, thread scheduling, timer firing, random entropy, and other relevant sources of nondeterminism. When feedback determinism holds while other nondeterminism remains, the concrete executions may still differ, but TAMIAS treats them as the same behavior because they produce the same feedback. Thus, the remaining nondeterminism does

Table 1. Target distributed systems. #GR denotes the number of grammar rules TAMIAS uses, excluding literal-only rules.

Name	Type	Lang.	LOC	Ver.	#GR
Redis [53]	KV store	C	327 K	7.4.1	104
MongoDB [59]	Doc database	C/C++	7268 K	8.0.1	56
MySQL [64]	SQL database	C/C++	5942 K	8.4.2	N/A ¹
etcd [23]	KV store	Golang	210 K	3.5.9	44
Nuraft [3]	Dist. protocol	C/C++	35 K	3.0.0	5
Braft [6]	Dist. protocol	C/C++	32 K	1.1.2	3

not change feedback-based decisions such as whether a test is retained or whether new feedback is discovered. For example, Redis’ CLIENT INFO command may report a different TCP port number across executions of the same test case, but always produces the same branch coverage. Therefore, the program behavior of this INFO command test case is treated as equivalent under the chosen branch-coverage feedback abstraction. When feedback determinism fails, the same retained test may instead produce different feedback across executions, making the corpus unstable: it may fail to reproduce the discovery for which it was retained. Feedback variations may also create redundant corpus entries: a mutation of an existing test may appear to discover new feedback solely because uncontrolled nondeterminism changes its execution, causing the fuzzer to retain it unnecessarily. Finally, TAMIAS’s current determinism guarantee is specific to branch-coverage feedback and does not automatically extend to finer-grained feedback or full execution determinism, which may expose additional nondeterminism that TAMIAS does not currently control.

4 Implementation

Implementation complexity. Table 5 in the appendix shows TAMIAS’s LOC breakdown and implementation languages. In total, TAMIAS comprises 13,496 lines of code.

Branch coverage collection. TAMIAS implements language-specific runtimes for collecting branch coverage. For C/C++ programs, TAMIAS uses LLVM’s trace-pc-guard instrumentation [68]. For Go programs, TAMIAS uses the `-d=libfuzzer` option [52], which records visited branches in an in-process bitmap. Upon receiving a signal, the TAMIAS Go runtime copies this bitmap to the fuzzer.

TAMIAS excludes coverage generated during system initialization because it is not input-driven. Marking such branches as covered could prevent later inputs that reach them from being identified as interesting. Therefore, TAMIAS clears coverage before taking the post-initialization VM snapshot (§3.6.1).

Bug detection. TAMIAS detects crashes, memory errors, and silent semantic bugs. It detects crashes by monitoring whether a node exits abnormally. For C/C++ targets, TAMIAS enables AddressSanitizer by default to detect memory errors.

¹For MySQL, #GR is not applicable because TAMIAS uses SQLSmith to generate input rather than a hand-written grammar.

Semantic checkers are target-specific. For etcd, we ported its CI semantic checkers, including its Porcupine-based linearizability checker [36] and watch-response checkers. Integrating them took about one person-day.

5 Evaluation

We evaluate TAMIAS’s effectiveness and performance on several large real-world distributed systems.

Testbed. We ran our experiments on bare-metal CloudLab machines [21] with two Intel Silver 4114 CPUs and 192 GiB of RAM, running Ubuntu 22.04 with hyper-threading enabled.

5.1 Target Applications

In this section, we discuss our experience in supporting new distributed systems on TAMIAS. We chose widely used real-world distributed systems, as shown in Table 1. For all systems, we tested their release versions. Thus, the evaluated targets have passed internal quality control and constitute a significantly higher bar for a testing tool than development versions do. The systems are implemented in different languages, namely C/C++ and Golang.

Configurations. We focus on testing the production-ready distributed modes of the target applications. For Redis, we test its cluster mode [54] using a three-node cluster, which shards the keys across the nodes. For etcd and MongoDB, we test their replication mode using a three-node cluster. For MySQL, we test its NDB cluster mode [63] using a five-node cluster, which consists of two SQL API nodes that serve client requests, an NDB management node, and two NDB data nodes. For Nuraft and Braft, we test their echo-server and atomic-register examples in three-node clusters.

Application modifications. TAMIAS requires no porting effort. We did not modify the source code of any target system. For MongoDB and Nuraft, we only disabled the `timerfd` option at compile time, because this feature is not yet supported by our current prototype. This does not require application changes: Both systems use ASIO [18], which provides a fallback timeout implementation based on `epoll_wait` when `timerfd` [45] is unavailable. We leave native `timerfd` support to future work.

5.2 Bug Finding

As shown in Table 2, TAMIAS found 13 distributed system bugs across five evaluated target systems. Twelve were previously unknown. These include crashes, memory errors, assertion failures, performance issues, and semantic bugs. Application developers have confirmed eight, including three that have been fixed. The bugs labeled as “analyzing” in the table are still under investigation due to the complexity of the target systems and the difficulty of reasoning about the bugs’ root causes. Among the bugs we analyzed, Redis-2, ETCD-2,

²The developer responded that the ETCD-2 bug had been reported previously; thus, we do not classify it as previously unknown.

Table 2. Bugs found by TAMIAS. Twelve bugs were previously unknown, and application developers have confirmed eight, including three that have been fixed.

ID	Description	Impact	Status	Introduced
Redis-1	Slot number assertion failure during reconfiguration	Crash	Pending	3.0.0 (2014)
Redis-2	Null pointer dereference of role structure during reconfiguration	Crash	Reported	3.0.0 (2014)
ETCD-1	Incorrect watch revision after snapshot transfer	Semantic	Fixed	3.1.11 (2017)
ETCD-2	Race between clearing and applying snapshots	Crash	Confirmed ²	3.5.0 (2020)
MySQL-1	Excessive epoll wakeups after a peer closes a connection	Performance	Confirmed	5.1.30 (2008)
MySQL-2	Index-statistics assertion failure during concurrent schema change	Crash	Confirmed	5.7.10 (2015)
MySQL-3	Schema change not propagated with an active remote client	Crash	Fixed	8.0.23 (2020)
MySQL-4	Assertion failure in <code>ha_ndbcluster::read_time</code>	Crash	Confirmed	8.0.28 (2021)
MySQL-5	Assertion failure in <code>ha_ndbcluster::get_metadata</code>	Crash	Analyzing	—
MySQL-6	Assertion failure in <code>Ndb_free_list_t()</code>	Crash	Analyzing	—
MySQL-7	Use-after-free during commit while blob operations are still in progress	ASAN	Analyzing	—
Mongo-1	Use-after-free when the oplog is truncated during a read	ASAN	Fixed	7.1.0 (2023)
Nuraft-1	Snapshot-finalization crash during a concurrent role change	Crash	Reported	3.0.0 (2023)

MySQL-2, MySQL-3, MySQL-4, Mongo-1, and Nuraft-1 require a specific ordering of nondeterministic events, such as reordered network messages or carefully constructed thread schedules. The remaining bugs can be triggered by specific sequences of client requests. MySQL-1 was found while we were configuring MySQL for TAMIAS. A more detailed analysis of all investigated bugs is provided in §A.1.

Case study: how TAMIAS triggers Nuraft-1. Here we study how TAMIAS’s mutation process helps trigger Nuraft-1 (Figure 2). TAMIAS tracks the mutation history and execution logs of each corpus test case, allowing us to inspect how a test case is generated and executed. We examine the mutation trace of a test case that triggers Nuraft-1. The mutation trace $62 \rightarrow 64 \rightarrow 70 \rightarrow 80 \rightarrow 82 \rightarrow 83 \rightarrow \text{bug}$ shows how the test case was mutated. Each number is a corpus test case ID, and each arrow denotes that one test case was derived from another through mutation. The first step from 62 to 64 mutates the input, while the later steps mainly mutate scheduler policies to explore new schedules from the retained prefix. By inspecting the execution logs, we found that the early corpus test cases (i.e., 62 and 64) first trigger an event-schedule prefix that grows the log and creates lagging followers, partially establishing ① in Figure 2. The subsequent corpus test cases (i.e., 70 and 80) preserve this prefix but explore other behaviors that do not contribute to the bug. Corpus test case 82 is the first to place the receiving node N1 into a long-lived snapshot-installation state, thus reaching ② on top of ①. Test case 83 drives the leadership change and the delivery of an overlapping second snapshot-installation message, reaching ③ and ④. Finally, the bug-triggering test case releases additional late requests only after this deep state has been reconstructed, triggering the crash at snapshot finalization, i.e., ⑤. A model checker could find this bug if given the correct input, but exploring the required schedule from scratch may

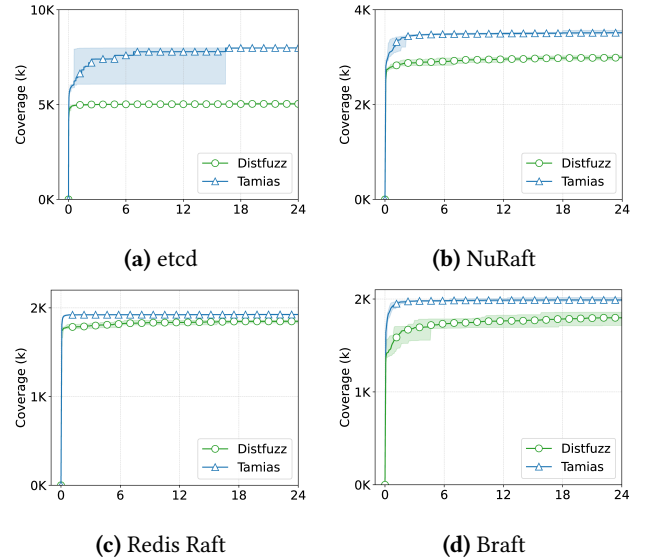


Figure 6. DistFuzz comparison over 24 hours.

still be inefficient. In our ablation, P-IM uses feedback-guided input mutation but runs PCTCP from scratch for each test; it did not find NuRaft-1 in ten 24-hour trials, whereas P-ISM additionally mutates event schedules using replay-explore and found the bug in two trials.

5.3 Comparison with Prior Work

In this section, we compare the effectiveness of TAMIAS with prior work. We choose DistFuzz [86] as the baseline because it is a recent state-of-the-art distributed system fuzzer. We use branch coverage as the evaluation metric because it is a basic measure of the behaviors explored through fuzzing.

Our goal is to compare the two systems’ effectiveness at exploring concurrency while minimizing differences in input grammars or fault models, which can significantly affect

Table 3. Branch coverage across TAMIAS configurations.

Target	Branch Coverage (Mean $\pm$ Std.)		
	R	R-IM	R-ISM (TAMIAS)
Redis	4,649 $\pm$ 24	5,124 $\pm$ 117	5,521 $\pm$ 52
MongoDB	90,812 $\pm$ 1,507	99,981 $\pm$ 5,156	110,997 $\pm$ 4,465
MySQL	21,481 $\pm$ 133	21,833 $\pm$ 221	22,416 $\pm$ 457
etcd	13,064 $\pm$ 849	13,296 $\pm$ 975	13,922 $\pm$ 76
Nuraft	3,420 $\pm$ 6	3,421 $\pm$ 4	3,423 $\pm$ 7
Braft	1,949 $\pm$ 6	1,968 $\pm$ 11	1,977 $\pm$ 8
	P	P-IM	P-ISM (TAMIAS)
Redis	4,809 $\pm$ 56	5,284 $\pm$ 149	5,442 $\pm$ 138
MongoDB	99,654 $\pm$ 4,494	110,276 $\pm$ 1,918	112,277 $\pm$ 856
MySQL	21,019 $\pm$ 97	21,072 $\pm$ 160	21,014 $\pm$ 178
etcd	13,783 $\pm$ 15	13,804 $\pm$ 23	13,952 $\pm$ 57
Nuraft	3,817 $\pm$ 34	3,868 $\pm$ 50	3,935 $\pm$ 79
Braft	1,973 $\pm$ 2	1,986 $\pm$ 3	1,986 $\pm$ 3

reachable code paths. To ensure a fair comparison, we align the two systems' external input spaces and test setups as closely as possible. Both fuzzers test the same instrumented binaries under the same configurations. DistFuzz supports a different fault model that may assume faulty OS or transport behavior, which is outside TAMIAS's target data-center deployment scenario. We therefore disable these additional faults. For example, DistFuzz injects message loss or duplication above TCP, which violates TCP's reliable byte-stream abstraction. We also disable node restart in DistFuzz, restrict TAMIAS to grammar elements supported by DistFuzz (e.g., SET and GET), and allocate five CPU cores to each system. For both systems, we clear the coverage bitmap after cluster initialization and measure only the coverage produced by each subsequent test execution.

We evaluate on the targets supported by both systems, namely etcd, NuRaft, Redis (Raft), and Braft. For each setup, we conduct 10 independent 24-hour trials. This 24-hour budget is common in prior fuzzing evaluations [42, 70].

Figure 6 shows the results. Compared to DistFuzz, TAMIAS achieves 47%, 19%, 4%, and 11% higher branch coverage on etcd, Nuraft, Redis (Raft), and Braft. This is mainly because TAMIAS can systematically steer executions toward diverse behaviors through fine-grained event control. For instance, TAMIAS triggered a legacy NuRaft snapshot-election bug [2] through systematic exploration of message deliveries and timer firings, exercising hundreds of additional error-handling branches, but DistFuzz did not trigger this bug in any of the 10 trials.

5.4 Ablation Study

We evaluate how the mutation of input and scheduler-policy programs contributes to fuzzing effectiveness. To control for randomness, we evaluate each configuration in 10 independent 24-hour trials, using 11 1-vCPU Nyx VMs per trial. We compare three configurations:

Table 4. Number of runs (out of 10) in which the bug was triggered in 24 hours. Bold indicates the best result within each scheduler algorithm.

Bug	Random Walk			PCTCP		
	R	R-IM	R-ISM (TAMIAS)	P	P-IM	P-ISM (TAMIAS)
Mongo-1	0	0	9	10	10	10
ETCD-1	0	1	2	0	2	4
MySQL-3	0	2	10	0	0	3
MySQL-4	4	4	5	0	0	0
Redis-1	0	9	10	0	8	8
Redis-2	0	0	5	0	0	6
Nuraft-1	0	0	0	0	0	2

- **Random exploration (R/P):** inputs and schedules are randomly generated without mutation; inputs are still derived from the grammar, and the scheduler-policy program contains a randomly generated policy that serves as the scheduler's initial configuration. R(-) and P(-) denote random-walk and PCTCP, respectively.
- **Input mutation (R-IM/P-IM):** random schedule exploration with input-program mutation;
- **Joint mutation (R-ISM/P-ISM):** mutation of both input and scheduler-policy programs.

5.4.1 Branch Coverage Effectiveness. In TAMIAS, branch coverage is measured using the number of unique branches executed in the target applications. Figure 7 and Figure 8 plot coverage over time; Table 3 reports final values.

For final branch coverage, as we gradually enable more mutation features, coverage generally improves across targets and scheduling algorithms. Compared with the baseline (R/P), enabling all features (R-ISM/P-ISM) improves coverage by 18.8%, 22.2%, 4.4%, and 6.6% for Redis, MongoDB, MySQL, and etcd, respectively, under the random-walk scheduler; and by 13.2%, 12.7%, and 1.2% for Redis, MongoDB, and etcd, respectively, under PCTCP. Nuraft also improves by 3.1% under PCTCP. While some of these relative improvements may seem modest, they still correspond to substantial absolute gains in coverage, ranging from tens to over twenty thousand additional covered branches depending on the target; for example, joint mutation covers 20,185 more branches in MongoDB under random walk and 12,623 more under PCTCP. Moreover, much of the baseline coverage comes from easily reachable code paths exercised by issuing simple valid client commands, which are triggered by the first few test cases and can be readily generated from our input grammar. Statistical significance is assessed using Mann-Whitney U-tests (detailed results in Appendix Table 6).

Interestingly, we also observe that the benefit of joint mutation appears related to the diversity of the input space. As shown in Table 1, several targets with larger grammars, such as Redis (#GR=104), MongoDB (#GR=56), and etcd (#GR=44), exhibit clearer gains from joint mutation, whereas targets with much smaller grammars, such as Nuraft (#GR=5) and

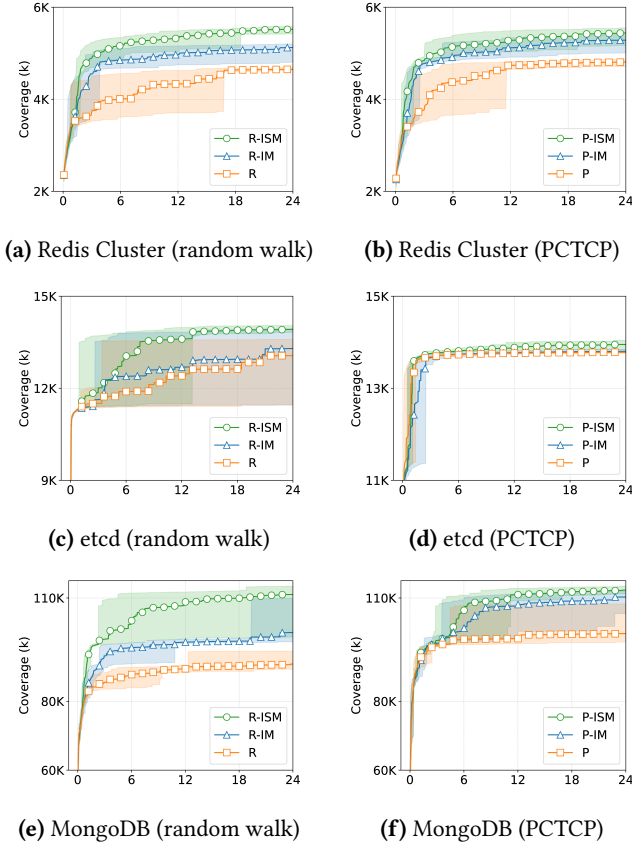


Figure 7. Branch coverage for Redis Cluster, etcd, and MongoDB. Lines show means; shading shows min–max ranges.

Braft (#GR=3), show more limited improvement. This is consistent with the intuition that joint mutation is most useful when the fuzzer can combine many input variants with event-schedule exploration, creating a larger search space.

Impact of scheduling algorithms. We observe that mutation in TAMIAS improves both evaluated scheduling algorithms, i.e., random walk and PCTCP, in branch coverage and bug finding. We also observe that, although PCTCP generally achieves higher coverage than random walk, it does not dominate in all cases. For example, on MySQL, random walk and its mutation variants (R-IM and R-ISM) achieve higher coverage than PCTCP and its variants, which may be partly due to PCTCP triggering more spinning behavior and thus limiting the benefit of additional mutation. This suggests that different scheduling algorithms may be better suited to different systems, and that TAMIAS’s mutation techniques can improve the effectiveness of multiple scheduling algorithms, providing flexibility in choosing the most suitable algorithm for a given target.

5.4.2 Bug Finding Effectiveness. We also report the effectiveness of our mutation techniques in triggering the previously unknown bugs. Table 4 shows the number of trials (out of 10) in which a bug was triggered within 24 hours. A

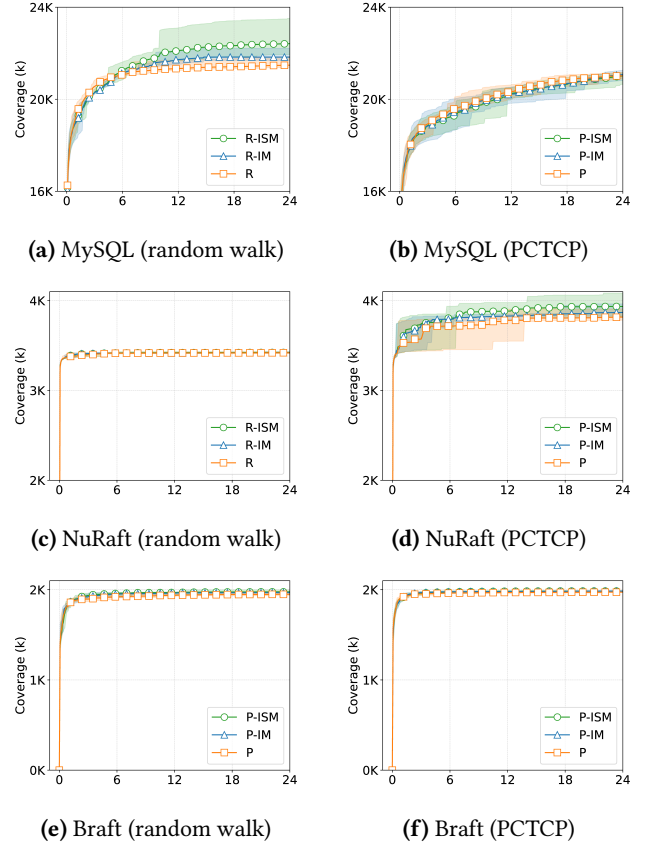


Figure 8. Branch coverage over time for MySQL, NuRaft, and Braft. Lines show means; shading shows min–max ranges.

bug is included if it was triggered by at least one configuration in at least one trial. Bugs not listed in this table require a larger testing budget to trigger. In general, mutation techniques improve bug-finding effectiveness. For example, the Redis-2 and Nuraft-1 bugs were only triggered when both input and schedule mutation were enabled (R-ISM/P-ISM).

5.5 Feedback Determinism

We evaluate feedback determinism by measuring the stability of each corpus test case. Specifically, we replay each corpus test case five times and compare the branch coverage. If a branch is not covered in all trials, we consider it unstable. All evaluated targets achieve feedback determinism.

6 Related Work

This section complements the related work in §2.

Feedback-guided distributed-system testing. Several recent systems apply feedback-guided testing to distributed systems but differ from TAMIAS in their targets and scope of control. ModelFuzz [33] uses coverage of states in an abstract formal model (e.g., a TLA+ specification) as feedback for testing distributed system implementations. This feedback captures protocol-level semantics but requires a suitable

formal model and an event-to-model mapping; its evaluated targets are also source-instrumented for message interception. DistFuzz [86] uses symmetry-pruned network-message sequences as black-box feedback, avoiding coverage instrumentation. TAMIAS instead uses branch coverage to guide mutation based on internal code-path behavior. §2 discusses how TAMIAS’s schedule exploration differs from DistFuzz.

UpFuzz [35] and Fest [44] are two recent concurrent works. UpFuzz targets data-format incompatibility bugs during distributed storage system upgrades, a problem complementary to fine-grained internal-event exploration. Fest generates schedules for distributed system models written in P, whereas TAMIAS tests unmodified implementations. Moreover, Fest focuses on schedule mutation, while TAMIAS jointly mutates inputs and fine-grained message, timer, and thread schedules. Finally, unlike Fest’s generic, sparse mutation of recorded random choices, TAMIAS uses scheduler-specific reconfiguration to preserve the scheduler’s suffix distribution, retaining the benefit of replay without weakening subsequent exploration (§3.5).

Model checking implementations. Much prior work has studied model checking for distributed system implementations. Such work often focuses on controlling and exploring nondeterministic events at fine granularity, enabling systematic exploration of the scheduling search space [43, 56, 81]. For example, SAMC [43] intercepts network messages from Java-based applications and delivers them one at a time. Some approaches [65, 81] further provide formal guarantees, such as sound reduction techniques to prune redundant schedules or sampling strategies with probabilistic guarantees. More recent systems [17, 74, 77] check whether a distributed system implementation conforms to a formally specified model (e.g., a TLA+ specification). However, existing approaches have several limitations. First, such techniques do not leverage runtime feedback to guide exploration. Second, they do not explore the external input space (e.g., client requests), instead relying on predefined input workloads provided by users. Finally, many require application changes [43, 56], adding porting effort. In addition, TAMIAS’s event-interception techniques differ from those of existing distributed system model checkers, as explained in Section 3.6.7. TAMIAS focuses on concurrency exploration and leaves disk and memory fault injection as future work.

Deterministic execution. Prior deterministic-execution and record-replay systems target threads, processes, containers, or kernels, rather than transparent distributed systems [8, 16, 51, 61, 62, 83]. ODR [4] relaxes the determinism requirement to output determinism for single-node multicore debugging to improve performance. FoundationDB [85] supports deterministic simulation testing but requires co-design of the DBMS and framework. Shadow [37, 38] executes unmodified distributed applications in a deterministic network simulator by intercepting all system calls [38]. TAMIAS instead

targets the fuzzing-sufficient guarantee of feedback determinism and shows that it can be supported through selective interposition, reducing interposition complexity and easing compatibility with large, real-world distributed systems. Moreover, Shadow does not provide feedback-guided fuzzing or schedule exploration. A recent concurrent work, Fizzle [7], provides deterministic coverage feedback for single-node network servers. However, Fizzle does not target multi-node distributed systems or integrate concurrency exploration algorithms with feedback-guided mutation. Furthermore, Fizzle relies on LD_PRELOAD-based library interposition, which cannot capture raw system calls that bypass dynamic libraries (e.g., those in the Go runtime), limiting its compatibility.

Finding distributed system bugs. Prior systems [12, 15, 20, 49, 50, 57, 69, 79, 84] target specific types of distributed system bugs. For example, CSnake [69] targets self-sustaining cascading distributed failures. These systems typically use domain-specific optimizations to prune the search space. In contrast, TAMIAS does not assume a particular bug class and provides a general testing framework for distributed systems.

Concurrency testing. Empirical studies have characterized real-world concurrency bugs in user-space applications and the Linux kernel [30, 55]. Other work develops testing techniques for concurrent programs and kernels [11, 14, 26, 28, 29, 31, 39, 40, 47, 60]. For example, Snowboard [29] uses potential memory communications inferred from dynamic memory-access traces to guide input and schedule exploration for kernel concurrency testing. Wolff et al. [78] use feedback-guided fuzzing to explore thread schedules in multithreaded programs. These approaches target single-host concurrency and do not control distributed-system events such as network-message delivery. TAMIAS instead jointly mutates client inputs and distributed event schedules spanning messages, timers, and threads.

7 Conclusion

In this paper, we presented TAMIAS, a distributed-system testing tool that combines transparent fine-grained event control with replay-explore mutation for systematic concurrency exploration. By targeting feedback determinism rather than full execution determinism, TAMIAS enables effective and practical testing without requiring broad system-level emulation or source-code modifications. Our evaluation on six real-world distributed systems shows that TAMIAS finds new bugs and improves coverage over the state of the art.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd, Peter Alvaro, for their insightful feedback. We are grateful to Sishuai Gong for suggestions on an earlier draft. We thank CloudLab for providing computing resources. This work was supported in part by gifts from Intel and Google and NSF awards CNS-2145888, CNS-2140305, and CNS-2300562.

References

- [1] AFL++ Team. 2026. Best Practices. https://aflplusplus/docs/best_practices/.
- [2] Jung-Sang Ahn. 2025. Make receiving_snapshot Flag Durable. <https://github.com/eBay/NuRaft/pull/590>. eBay/NuRaft pull request #590.
- [3] Jung-Sang Ahn. 2026. eBay/Nuraft - C++ implementation of Raft core logic as a replication library. <https://github.com/eBay/NuRaft>.
- [4] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *ACM Symposium on Operating Systems Principles (SOSP '09)*. doi:10.1145/1629575.1629594
- [5] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars. In *Network and Distributed System Security Symposium (NDSS '19)*. doi:10.14722/ndss.2019.23412
- [6] Baidu, Inc. 2026. braft: An industrial-grade C++ implementation of RAFT consensus algorithm based on brpc. <https://github.com/baidu/braft>.
- [7] Nathaniel Bennett, Tyler Tucker, Carson Stillman, William Enck, Patrick Traynor, and Kevin Butler. 2026. Fizzle: A Framework for Deterministic and Reproducible Network Fuzzing. In *IEEE Symposium on Security and Privacy (S&P '26)*. doi:10.1109/SP63933.2026.00091
- [8] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*. <https://www.usenix.org/conference/osdi10/deterministic-process-groups-dos>
- [9] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. doi:10.1145/2976749.2978428
- [10] Martin A. Brown. 2007. Guide to IP Layer Network Administration with Linux. <http://linux-ip.net/html/routing-tables.html#routing-table-local>.
- [11] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*. doi:10.1145/1736020.1736040
- [12] Hongchen Cao, Jingzhu He, Ting Dai, and Guoliang Jin. 2026. Themis: Detecting Distributed Concurrency Bugs through RPC-Driven Race-Directed Test Generation and Fuzzing. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '26)*. Renton, WA. <https://www.usenix.org/conference/nsdi26/presentation/cao>
- [13] Tingjia Cao, Shawn (Wanxiang) Zhong, Caeden Whitaker, Ke Han, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2026. kSTEP: Characterization and Deterministic Testing of Linux CPU Scheduler Bugs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '26)*. <https://www.usenix.org/conference/osdi26/presentation/cao>
- [14] 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 *USENIX Security Symposium (USENIX Security '20)*. <https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu>
- [15] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Ming Gu, Qing Liao, and Yu Jiang. 2024. Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay. In *IEEE Symposium on Security and Privacy (S&P '24)*. doi:10.1109/SP54263.2024.00109
- [16] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. 2011. Efficient deterministic multithreading through schedule relaxation. In *ACM Symposium on Operating Systems Principles (SOSP '11)*. doi:10.1145/2043556.2043588
- [17] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. 2020. Extreme modelling in practice. *Proc. VLDB Endow.* 13, 9 (2020), 1346–1358. doi:10.14778/3397230.3397233
- [18] Asio developers. 2026. The Asio C++ library. https://think-async.com/Asio/boost_asio_1_30_2/doc/html/boost_asio.html.
- [19] Google developers. 2026. syzkaller - kernel fuzzer. <https://github.com/google/syzkaller>.
- [20] Gen Dong, Yu Hua, Yongle Zhang, Zhangyu Chen, and Menglei Chen. 2025. Understanding and Detecting Fail-Slow Hardware Failure Bugs in Cloud Systems. In *USENIX Annual Technical Conference (USENIX ATC '25)*. <https://www.usenix.org/conference/atc25/presentation/dong>
- [21] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *USENIX Annual Technical Conference (USENIX ATC '19)*. <https://www.usenix.org/conference/atc19/presentation/duplyakin>
- [22] eBPF.io authors. 2026. eBPF Documentation. <https://ebpf.io/what-is-ebpf/>.
- [23] etcd Authors. 2026. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. <https://etcd.io/>.
- [24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies (WOOT '20)*. <https://www.usenix.org/conference/woot20/presentation/fioraldi>
- [25] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. doi:10.1145/3548606.3560602
- [26] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/fonseca>
- [27] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *European Conference on Computer Systems (EuroSys '17)*. 328–343. doi:10.1145/3064176.3064183
- [28] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. 2023. Probabilistic Concurrency Testing for Weak Memory Programs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. doi:10.1145/3575693.3575729
- [29] Sishuai Gong, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *ACM Symposium on Operating Systems Principles (SOSP '21)*. doi:10.1145/3477132.3483549
- [30] Sishuai Gong, Chih-En Lin, Kevin Wu, Edwin Lu, and Pedro Fonseca. 2026. A Comprehensive Study of Concurrency Bugs in the Linux Kernel. In *IEEE/ACM International Conference on Software Engineering (ICSE '26)*.
- [31] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. 2023. Snowcat: Efficient Kernel Concurrency Testing using a Learned Coverage Predictor. In *ACM Symposium on Operating Systems Principles (SOSP '23)*. doi:10.1145/3600006.3613148
- [32] Google. 2026. Good Fuzz Target. <https://github.com/google/fuzzing/blob/master/docs/good-fuzz-target.md>.
- [33] Ege Berkay Gulcan, Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Srinidhi Nagendra. 2025. Model-Guided Fuzzing of Distributed Systems. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 282 (Oct. 2025), 28 pages. doi:10.1145/3763060

- [34] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex C. Snoeren, and Geoffrey M. Voelker. 2011. DieCast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS)* 29, 2 (2011), 4:1–4:48. doi:10.1145/1963559.1963560
- [35] Ke Han, Sruthi P C, Yayu Wang, Yaoxu Song, Bishal Basak Papan, Junwen Yang, Pedro Fonseca, and Yongle Zhang. 2026. UpFuzz: Detecting Data Format Incompatibility Bugs during Distributed Storage System Upgrade. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '26)*. <https://www.usenix.org/conference/nsdi26/presentation/han>
- [36] Alex Horn and Daniel Kroening. 2015. Faster Linearizability Checking via P-Compositionality. arXiv preprint arXiv:1504.00204. <https://arxiv.org/abs/1504.00204>
- [37] Rob Jansen and Nicholas Hopper. 2012. Shadow: Running Tor in a Box for Accurate and Efficient Experimentation. In *Network and Distributed System Security Symposium (NDSS '12)*. <https://www.ndss-symposium.org/ndss2012/ndss-2012-programme/shadow-running-tor-box-accurate-and-efficient-experimentation/>
- [38] Rob Jansen, Jim Newsome, and Ryan Wails. 2022. Co-opting Linux Processes for High-Performance Network Simulation. In *USENIX Annual Technical Conference (USENIX ATC '22)*. <https://www.usenix.org/conference/atc22/presentation/jansen>
- [39] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzler: Finding Kernel Race Bugs through Fuzzing. In *IEEE Symposium on Security and Privacy (S&P '19)*. doi:10.1109/SP.2019.00017
- [40] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In *Network and Distributed System Security Symposium (NDSS '22)*.
- [41] Kyle Kingsbury. 2026. Distributed Systems Safety Research. <https://jepsen.io/>.
- [42] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. doi:10.1145/324373.4.3243804
- [43] 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/leesatapornwongsa>
- [44] Ao Li, Ankush Desai, and Rohan Padhye. 2026. Feedback-guided Adaptive Testing of Distributed Systems Designs. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '26)*. <https://www.usenix.org/conference/nsdi26/presentation/li>
- [45] Linux man-pages project. 2026. timerfd_create(2) – Linux manual page. https://man7.org/linux/man-pages/man2/timerfd_create.2.html
- [46] Linux man-pages project. 2026. The vDSO Library. <https://man7.org/linux/man-pages/man7/vdso.7.html>
- [47] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '23)*. doi:10.1145/3575693.3575731
- [48] Congyu Liu, Yaoxu Song, Yongle Zhang, and Pedro Fonseca. 2026. Tamias Artifact. <https://github.com/rssys/tamias-artifact>
- [49] Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. 2017. DCatch: Automatically Detecting Distributed Concurrency Bugs in Cloud Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. doi:10.1145/3037697.3037735
- [50] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. doi:10.1145/3173162.3177161
- [51] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: efficient deterministic multithreading. In *ACM Symposium on Operating Systems Principles (SOSP '11)*. doi:10.1145/2043556.2043587
- [52] LLVM Project. 2026. libFuzzer – a library for coverage-guided fuzz testing. <https://llvm.org/docs/LibFuzzer.html>
- [53] Redis Ltd. 2026. Redis. <https://redis.io/docs/latest/>
- [54] Redis Ltd. 2026. Redis cluster mode. https://redis.io/docs/latest/operate/oss_and_stack/management/scaling/
- [55] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuan Yuan Zhou. 2008. Learning from Mistakes: A Comprehensive Study on Real-World Concurrency Bug Characteristics. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*. doi:10.1145/1346281.1346323
- [56] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *European Conference on Computer Systems (EuroSys '19)*. Article 20. doi:10.1145/3302424.3303986
- [57] Tao Lyu, Liyi Zhang, Zhiyao Feng, Yueyang Pan, Yujie Ren, Meng Xu, Mathias Payer, and Sanidhya Kashyap. 2024. Monarch: A Fuzzing Framework for Distributed File Systems. In *USENIX Annual Technical Conference (USENIX ATC '24)*. <https://www.usenix.org/conference/atc24/presentation/lyu>
- [58] Ruijie Meng, George Pirlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox Fuzzing of Distributed Systems. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. doi:10.1145/3576915.3623097
- [59] MongoDB, Inc. 2026. MongoDB: The Developer Data Platform | MongoDB. <https://www.mongodb.com/docs/>
- [60] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, P. Ramanayagam Arumuga Nainar, and Iulian Neamtii. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*. <https://www.usenix.org/conference/osdi-08/finding-and-reproducing-heisenbugs-concurrent-programs>
- [61] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. doi:10.1145/3373376.3378519
- [62] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability. In *USENIX Annual Technical Conference (USENIX ATC '17)*. <https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan>
- [63] Oracle Corporation. 2026. MySQL NDB cluster mode. <https://www.mysql.com/products/cluster/>
- [64] Oracle Corporation. 2026. MySQL Reference Manual. <https://dev.mysql.com/doc/>
- [65] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. 2018. Randomized testing of distributed systems with probabilistic guarantees. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 160 (2018), 28 pages. doi:10.1145/3276530
- [66] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *IEEE International Conference on Software Testing, Validation and Verification (ICST '20)*. doi:10.1109/ICST46399.2020.00062
- [67] PingCAP-QE. 2026. go-sqlsmith is a port of sqlsmith. <https://github.com/PingCAP-QE/go-sqlsmith>

- [68] LLVM project developers. 2026. SanitizerCoverage - LLVM 22.0.0git documentation. <https://clang.llvm.org/docs/SanitizerCoverage.html>.
- [69] Shangshu Qian, Lin Tan, and Yongle Zhang. 2026. CSnake: Detecting Self-Sustaining Cascading Failure via Causal Stitching of Fault Propagations. In *European Conference on Computer Systems (EuroSys '26)*. doi:10.1145/3767295.3769373
- [70] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P '24)*. doi:10.1109/SP54263.2024.00137
- [71] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. 2021. Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types. In *USENIX Security Symposium (USENIX Security '21)*. <https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo>
- [72] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-Net: Network Fuzzing with Incremental Snapshots. In *European Conference on Computer Systems (EuroSys '22)*. doi:10.1145/3492321.3519591
- [73] Qingkai Shi, Junyang Shao, Yapeng Ye, Mingwei Zheng, and Xiangyu Zhang. 2023. Lifting Network Protocol Implementation to Precise Format Specification with Security Applications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS '23)*. doi:10.1145/3576915.3616614
- [74] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. 2024. SandTable: Scalable Distributed System Model Checking with Specification-Level State Exploration. In *European Conference on Computer Systems (EuroSys '24)*. doi:10.1145/3627703.3650077
- [75] The Go Authors. 2026. Go Fuzzing. <https://go.dev/doc/security/fuzz/>.
- [76] The Linux Kernel Developers. 2019. Seccomp BPF. https://www.kernel.org/doc/html/v5.1/userspace-api/seccomp_filter.html.
- [77] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *European Conference on Computer Systems (EuroSys '23)*. doi:10.1145/3552326.3587442
- [78] Dylan Wolff, Zheng Shi, Gregory J. Duck, Umang Mathur, and Abhik Roychoudhury. 2024. Greybox Fuzzing for Concurrency Testing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '24)*. doi:10.1145/3620665.3640389
- [79] Haoze Wu, Jia Pan, and Peng Huang. 2024. Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI '24)*. <https://www.usenix.org/conference/nsdi24/presentation/wu-haoze>
- [80] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. 2022. DocTer: documentation-guided fuzzing for testing deep learning API functions. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22)*. doi:10.1145/3533767.3534220
- [81] 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 *USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*. <https://www.usenix.org/conference/nsdi-09/modist-transparent-model-checking-unmodified-distributed-systems>
- [82] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial Order Aware Concurrency Sampling. In *International Conference on Computer Aided Verification (CAV '18)*. doi:10.1007/978-3-319-96142-2_20
- [83] Tianren Zhang, Sishuai Gong, and Pedro Fonseca. 2025. KRR: Efficient and Scalable Kernel Record Replay. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI '25)*. <https://www.usenix.org/conference/osdi25/presentation/zhang-tianren>
- [84] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and Detecting Software Upgrade Failures in Distributed Systems. In *ACM Symposium on Operating Systems Principles (SOSP '21)*. doi:10.1145/3477132.3483577
- [85] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '21)*. doi:10.1145/3448016.3457559
- [86] Yonghao Zou, Jia-Ju Bai, Zu-Ming Jiang, Ming Zhao, and Diyu Zhou. 2025. Blackbox Fuzzing of Distributed Systems with Multi-Dimensional Inputs and Symmetry-Based Feedback Pruning. In *Network and Distributed System Security Symposium (NDSS '25)*. doi:10.14722/ndss.2025.241912

A Appendix (Not Peer-Reviewed)

A.1 New Bug Descriptions

This section describes in detail the symptoms and root causes of the new bugs found by TAMIAS.

Redis-1. In a Redis cluster, each key is assigned to a slot, and each master owns a partition of the slots. By design, a replica node is not allowed to hold slots. However, the ADDSLOT command can add new slots to a replica node. Later, changing this replica node’s master triggers an assertion failure.

Redis-2. In a Redis cluster, a node stores its role in flag, e.g., whether it is a master or replica. When it is a replica node, it also uses a slaveof pointer to track its master node metadata. When the flag is set to replica, the slaveof pointer should not be NULL. However, after a replica node is reconfigured as a master, if it receives a stale MEET message that it sent to itself while it was a replica, the node incorrectly sets its flag to replica while leaving slaveof as NULL. This triggers a NULL pointer dereference because some code assumes that slaveof is non-NULL for a replica node.

ETCD-1. ETCD uses Raft as its state machine replication protocol. ETCD maintains a revision, which is a global version counter that increases monotonically whenever key-value changes occur. ETCD allows a client to watch for changes to certain key ranges starting from a given revision. However, when a follower node lags behind (e.g., due to a transient network partition) and receives a snapshot from the leader, it incorrectly decreases the user-specified starting revision of ongoing watch requests. Clients then receive watch responses with revisions lower than the specified starting revision.

ETCD-2. During a leader change, the new leader produced two snapshots in rapid succession (e.g., due to a network partition). The follower began applying the first snapshot, but before completion, the arrival of the next snapshot caused the cleanup logic to delete the earlier snapshot as “orphaned”. When the follower later attempted to open the backend for that snapshot, the file was missing, leading to a panic.

MySQL-1. MySQL cluster nodes use `epoll_wait` to monitor network connections. Specifically, each node uses one thread to monitor `epoll` events and another thread that wakes up periodically (every 100 ms) to garbage-collect closed connections. The monitoring thread informs the garbage collection thread by updating the shared data structures. Because the monitoring thread does not remove the closed socket from the `epoll` file descriptor, the thread is immediately woken again when it enters `epoll_wait` (due to the level-triggered mode), incurring high CPU usage. This continues until the garbage-collection thread wakes up and removes the closed socket from the `epoll` file descriptor.

Table 5. Core components of TAMIAS’s implementation.

Component	LOC	Language
Fuzzer	3,152	Go
Scheduler		
Scheduler misc.	2,678	Go
Network proxy	420	Go
Monitor (userspace)	2,365	Go
Monitor (eBPF)	1,343	C
In-process runtime	3,538	C
Total	13,496	

We discovered this performance bug while configuring MySQL for TAMIAS: allowing longer execution before scheduler preemption exposed prolonged spinning caused by these repeated wakeups.

MySQL-2. A typical MySQL cluster consists of three types of nodes: management nodes (`ndb_mgmd`), data nodes (`ndbd`), and API nodes (`mysqld`). API nodes receive SQL requests from clients and interact with the data nodes to process them. This bug causes the assertion `index_stat_list==nullptr` to fail on API nodes, leading to crashes. The bug occurs when an API node accesses index statistics during query processing after receiving a schema update event. Specifically, the API node uses `NDB_SHARE` as a reference to a table-metadata structure, and the `index_stat_list` field in `NDB_SHARE` tracks the allocated table index-statistics structures used to optimize queries. The assertion requires `index_stat_list` to be freed before a dropped `NDB_SHARE` structure is destroyed. However, this assertion is violated when the following conditions are met: An API node receives a SQL update request and starts processing it. Later, before processing finishes, this node receives a schema update event for the same table from a data node, which can be caused by an `ALTER` request from another API node. A thread starts invalidating the stale table metadata and also marks the `NDB_SHARE` used by the SQL-processing thread as dropped. If the SQL-processing thread subsequently allocates and retrieves an index-statistics structure, `index_stat_list` becomes non-NULL, thereby violating the aforementioned assertion.

MySQL-3. In MySQL NDB with two API nodes, creating a secondary index can race with table-share caching. When an API node handles an index schema update, it invalidates the table cache; however, a stale table share without the new index can be recreated immediately after invalidation and escape further invalidation. If another thread later processes an `ALTER TABLE` command using this stale share, the NDB metadata and the DD table definition diverge, triggering an assertion failure.

MySQL-4. In MySQL NDB, each API node maintains table statistics used by the query optimizer. The statistics field `stats.records` must always be non-negative. However, after a concurrent insert and transactional delete+update, the

Table 6. Mann-Whitney U-test P-values comparing the effectiveness of different test case generation methods. Bold values indicate statistical significance at the 0.05 level.

	Redis	MongoDB	MySQL	etcd	Nuraft	Braft
R/R-IM	1.8e-04	3.3e-04	1.2e-02	2.1e-02	2.4e-01	5.6e-03
R-IM/R-ISM	1.8e-04	1.8e-04	8.5e-03	7.3e-03	4.0e-01	7.5e-02
R/R-ISM	1.8e-04	1.8e-04	1.8e-04	1.8e-04	7.3e-01	1.8e-04
P/P-IM	1.8e-04	3.3e-04	5.1e-2	1.8e-04	2.2e-03	1.6e-04
P-IM/P-ISM	1.1e-02	5.8e-03	5.7e-01	2.4e-04	2.8e-02	7.9e-01
P/P-ISM	1.8e-04	3.3e-04	8.5e-01	1.8e-04	5.8e-04	1.7e-04

stats.records value can be updated to -1 (interpreted as UINT_MAX), leading to an assertion failure.

Mongo-1. In MongoDB, a use-after-free can occur during collection scanning when the oplog is resized concurrently. The collection scan stage explicitly abandons the active snapshot if it needs to wait for earlier oplog writes. If a concurrent resize operation frees and replaces the oplog instance during this window, the existing CollectionPtr becomes invalid. Subsequent accesses in the scan continue to use the stale pointer, resulting in a heap use-after-free and server crash. Although the possibility of invalidating a CollectionPtr during snapshot abandonment had previously been mentioned in the issue tracker, no concrete triggering scenario was known. Our testing identified a specific workload that reliably triggers a use-after-free, and we provided ASAN evidence to developers. They confirmed and released a fix soon after our report, indicating that our finding exposed a previously unconfirmed bug.

A.2 TAMIAS Performance

In this section, we measure TAMIAS’s fuzzing throughput on different targets. TAMIAS executes test cases using a 1-vCPU Nyx VM [71], which supports fast VM snapshot reloading. Thus, TAMIAS can easily scale the execution throughput by increasing the number of VMs it uses. To measure peak performance, we disable TAMIAS’s auxiliary diagnostic features, such as the collection of node logs and working directories. The virtual time budget for each execution is set to 1 second. We conduct the experiments on a machine with a 24-core AMD EPYC 7402P processor (48 hardware threads) and 128 GiB of RAM. TAMIAS is configured to use 36 VMs. We run each fuzzing campaign for 5 minutes.

As shown in Table 7, fuzzing Redis achieves the highest throughput, up to 33 executions per second, while the other target systems have lower throughput. Two factors explain these results. First, these systems produce more events than Redis. For example, MySQL (NDB cluster) produces more timer events through `epoll_wait` (every 10 milliseconds), which adds scheduling overhead for TAMIAS. Second, some target-system clusters contain more nodes. For example, we use 5 nodes to build the MySQL cluster and test its sharding and replication functions.

Table 7. Fuzzing throughput (executions/s) using 36 1-vCPU VMs on a machine with 24 cores (48 hardware threads).

Redis	ETCD	MongoDB	MySQL	Nuraft	Braft
33	4	7	4	24	28

TAMIAS achieves higher throughput than distributed-system testing tools that systematically control event schedules. For example, other tools [43] report exercising more than 100,000 executions in 48 machine-days, which corresponds to much lower throughput than TAMIAS provides.

A.3 Applying the Schedule-Control Policy

Algorithm 4 details the ApplyPolicy call in Algorithm 2: it processes the command list $Q = P[s]$ before Run selects the next event.

Algorithm 4 APPLYPOLICY(A, I, clk, Q).

Require: Algorithm A , input program I , virtual clock clk , policy-command list Q

```

1: for all  $p \in Q$  do
2:   if  $p.type = \text{ALGORITHM\_POLICY}$  then
3:      $A.RECONFIG(p.args)$   $\triangleright$  reconfigure future choices
4:   else if  $p.type = \text{INPUT\_POLICY}$  then
5:      $E \leftarrow \text{TOEVENTS}(I[p.ids])$   $\triangleright$  resolve referenced inputs
6:      $A.ADDEVENTS(E)$   $\triangleright$  make inputs schedulable
7:   else if  $p.type = \text{VIRTUAL\_TIME\_POLICY}$  then
8:      $clk.timerProb \leftarrow p.probability$   $\triangleright$  update timer selection
9:   end if
10: end for

```