Rose: Reproducing External-Fault-Induced Failures in Distributed Systems with Lightweight Instrumentation

Sebastião Amaro
IST Lisbon & INESC-ID
Lisbon, Portugal

Pedro Fonseca
Purdue University
West Lafayette, Indiana, U.S.A

Miguel Matos
IST Lisbon & INESC-ID
Lisbon, Portugal

Abstract

Distributed systems form the backbone of critical infrastructures, yet remain vulnerable to *external-fault-induced bugs* that manifest only when specific external events occur during specific application states. Existing approaches to reproducing these bugs require fine-grained information about the application, which might not be available in production, and operate within a limited fault model. Rose is a novel approach that collects traces from production and systematically generates fault schedules that reproduce these bugs. By leveraging the insight that external faults are observable through system interfaces, Rose uses lightweight tracing (2.6% overhead) to capture essential application-environment interactions. Then, it identifies the application states when faults must occur to trigger bugs, and generates schedules that consistently reproduce these bugs. Rose reproduced 20 bugs across eight production systems implemented in diverse languages (C, C++, Java, Go, Scala), including widely-used systems such as Zookeeper, MongoDB, and HBase.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; • **Computer systems organization → Reliability**.

Keywords: Bug Reproduction, Fault Injection, Distributed Systems

ACM Reference Format:
Sebastião Amaro, Pedro Fonseca, and Miguel Matos. 2026. Rose: Reproducing External-Fault-Induced Failures in Distributed Systems with Lightweight Instrumentation. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3767295.3803625

1 Introduction

Distributed systems underpin critical infrastructure services, and their failures can result in substantial revenue losses and service disruptions [1–3]. Despite advances in, and adoption of, formally proven algorithms [4] and fault tolerance mechanisms [5, 6], widely used distributed system implementations remain vulnerable to failures due to software bugs [7] that incorrectly handle hardware and network faults [8, 9].

In this paper we focus on *external-fault-induced bugs* (EFIBs) — software defects that manifest only when unexpected external events, such as network partitions, disk errors or OS errors, trigger erroneous program states that lead to failures. Unlike typical bugs that are triggered with specific user input, EFIBs often involve defects in the error handling logic of distributed systems, which is particularly complex, hard to validate in deployment, and difficult to test during development due to the myriad of corner cases.

External faults are outside of the program control. Systems employ mechanisms to gracefully handle these faults and continue their normal execution correctly. However, when these faults occur in a specific order and/or in a specific system state they might reveal implementation defects — we call these defects EFIBs. These factors make analyzing and understanding these bugs extremely hard, and few techniques exist to help developers fix them.

Several tools such as Jepsen [10], Mallory [11], PACE [12], Chronos [13], Sieve [14], FCatch [15], and CrashMonkey [16] employ various strategies to discover bugs by injecting faults and observing system behavior. Nonetheless, in practice these techniques can not find all bugs due to the massive faults and input search space, and hence many critical bugs still find their way into production leading to failures.

Thus, the ability to reproduce production failures is key to identifying and fixing these bugs. Unfortunately, this is a complex, time-consuming, challenging process for developers, that often involves manual and ad-hoc approaches, such as manual inspection of logs and code with repeated experimentation of possible fixes. As previous work has shown, *"developers spend a vast majority of the resolution time (69%) on reproducing the failure"* [17].

Bug finding and reproduction are two distinct tasks, with their specific challenges. Whereas bug finding typically involves exploring a large fault space to uncover potential issues, failure reproduction requires precisely recreating the specific *context* that triggered a bug. The challenge in reproducing EFIBs is particularly significant for production failures, because tight overhead constraints in production systems limit the information that can be recorded and used,

EUROSYS '26, Edinburgh, Scotland Uk

ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3803625

and the complex interplay between system state and fault timing makes failure reproduction extremely difficult.

State-of-the-art works leverage system-specific internals to reproduce fault-induced failures in testing environments. Pensieve [17] reconstructs the inputs and steps that led to a failure from log files, while Anduril [18] mimics the occurrence of faults by injecting exceptions at specific points of the execution. While effective, they are limited to JVM-based deployments since they leverage specifics about Java and the JVM to direct their approaches. Furthermore, they require fine-grained logs which might not always be available.

The tension between available information and failure reproduction in distributed systems creates a fundamental challenge. Collecting comprehensive runtime information simplifies bug reproduction but entails high overheads for production environments. As an example, record and replay approaches [19–22] capture all non-deterministic behavior, such as program inputs and thread interleavings. While these can help reproduce EFIBs during testing runs, they are inapplicable in production given their prohibitive overheads (e.g., RR [22] has a minimum overhead of 1.49×).

To address this challenge, we leverage the insight that only a subset of environment information is often relevant for a given failure. Furthermore, key interactions between the program and the environment can be recorded with low overhead, potentially allowing the inference of sufficient context to inject the faults that induce EFIBs.

To this end, this paper introduces Rose a novel approach that automatically reproduces EFIBs in diverse distributed systems without application-specific annotations or knowledge. Since the interactions with the environment typically happen through well-defined system interfaces (primarily system calls) Rose strategically monitors system call boundaries to capture the essential contextual information required for bug reproduction with minimal overhead.

Realizing this approach requires addressing two key research challenges. First, given a system trace leading to a bug, we must find *what* faults occurred, and *when* they happened relative to the system execution. For instance, certain faults are mostly innocuous unless they happen when the system is executing specific functions (see §3 for a concrete motivating example). This requires distinguishing between normal operation failures (which are expected and handled) and the specific fault sequence that triggered the bug. We identify common and anomalous events by comparing the production trace of the buggy execution against the trace of a fault-free testing run and, through a diagnosis phase, determine the precise conditions necessary for bug reproduction.

The second challenge involves reproducing production failures in a testing environment. After identifying the context that potentially leads to a fault, we need *precise fault injection* mechanisms to inject faults at exact points in the system execution. Without precision, attempts at reproduction would yield inconsistent results, making the process of finding the correct context impossible. To solve this challenge, we leverage the insight that external faults only manifest in system behavior at well-defined externalization calls, specifically system calls and network communications. By injecting faults precisely at these externalization points, we have the control needed for consistent bug reproduction without language-specific instrumentation.

Rose implements this approach using eBPF [23] to efficiently and safely observe the application, without requiring kernel changes that could hinder production deployments.

In our evaluation Rose automatically reproduced 20 EFIBs across 8 widely-used production systems written in different languages, namely HBase (Java), HDFS (Java), Kafka (Java/Scala), MongoDB (C++), RedisRaft (C), Redpanda (C++), Tendermint (Go), and Zookeeper (Java). We show that Rose has an overhead of 2.6% by collecting only partial information about the system and its interactions with the environment. Finally, we present a discussion on the reproduced bugs, where we delve into what key observations Rose makes to reproduce each bug.

Contributions. This paper makes the following contributions:

- A novel method to reproduce EFIBs in distributed systems.
- The design and implementation of Rose, which has three components. A lightweight tracer designed for production environments that captures crucial externally observable behavior with low overhead per node. An analyzer that automatically creates precise fault schedules that reliably reproduce bugs. An executor which provides the necessary infrastructure to run schedules in a testing environment.
- An extensive evaluation demonstrating Rose effectiveness in automatically reproducing 20 EFIBs across 8 widely-used systems.

2 Background

This section provides an overview of the key technologies underpinning our approach, specifically how we record application interactions with the environment, and how we precisely inject faults. To this end, we leverage several features of the Linux kernel.

eBPF (extended Berkeley Packet Filter) eBPF enables the execution of custom programs within the Linux kernel in a sandboxed environment without kernel code modifications. eBPF programs run with high performance (by executing directly in the kernel) and safety (through verification mechanisms that ensure isolation and termination). We note that the restrictions imposed by the eBPF verifier, such as checking constraining memory accesses, preventing deadlocks, restricting the number of instructions, among others, make developing these programs potentially challenging.

In our approach, eBPF serves two critical purposes:

- Fault injection: using `bpf_override_return` [24] to manipulate system call results and `bpf_send_signal` [25] to control (e.g.: pause, crash) processes from kernelspace.
- System tracing: monitoring both kernel activity (through `tracepoints` [26] and syscall `probes` [27]) and application behavior (via user-function probes [28]).

XDP (eXpress Data Path) XDP complements our tracing capabilities by providing a high-performance framework for processing network packets at the earliest point in the networking stack, right after a packet is received by a network interface card (NIC), and before it is passed to the kernel's networking stack. We use XDP to efficiently monitor network traffic with minimal overhead, capturing communication patterns essential for reproducing network-related faults. Some NICs allow running XDP programs entirely on the NIC, allowing users to offload their work from the kernel.
TC (Traffic Control) TC allows precise control over packet transmission, including selectively dropping packets to emulate network partitions and failures. While XDP operates only at packet ingress, TC works at ingress and egress points, enabling us to create diverse fault scenarios.

3 A Reproducibility Challenge

To demonstrate the difficulty in reproducing EFIBs, let us consider bug `RedisRaft-43` [29] (detailed in Table 1 and §6), a real bug discovered by Jepsen [10] in RedisRaft [30], a Redis module implementing the Raft consensus algorithm. This bug manifests as a node panicking on restart, due to a mismatch between log and snapshot indexes. Jepsen discovered this bug by randomly injecting a combination of process crashes, pauses, and network partitions. The only information reported is an error message from a failed assertion that validates these indexes must match.

As a preliminary attempt to reproduce this bug, we analyzed the Jepsen test history to extract the sequence of faults injected right before the crash. Next, we manually created a simple schedule incorporating these faults in an attempt to trigger the bug. We empirically found that the last three faults (we tried other combinations) before the crash were enough to reproduce the bug, but with a very low success rate (1 in 100 executions), demonstrating the difficulty of manually recreating the precise conditions needed. Some bugs are even more difficult to analyze, which explains why some remain unfixed, despite having been triggered in production, for days, weeks, or even months. Pensieve [17] reports that the average absolute time until a failure is reproduced is around 79 days (69% of resolution time). This highlights two key challenges. First, random fault injection provides no guarantee the bug will recur in subsequent test runs, making timely debugging impossible. Second, error messages alone provide insufficient information to determine *what* fault conditions triggered the bug or *when* they occurred relative to the system state. Such a low replay rate and scarce information makes it hard for developers to understand the causes of the bug, and time-consuming to test potential fixes.

When we used Rose to analyze the trace from the single successful reproduction, it automatically identified the three critical faults and the specific context in which they needed to occur, generating a schedule that reproduced the bug with 100% replay rate. The key insight was that the third fault had to happen when a node was executing a specific function.

4 Rose

Rose is designed to help developers reproduce EFIBs that previously occurred in production systems.

Definition 4.1. An external-fault-induced bug (EFIB) is a software defect that satisfies two conditions:

- Event Externality: The bug is triggered by environmental events $E = \{e_1, \ldots, e_n\}$ outside program control (system call failures, network delays, process crashes),
- Context Dependency: The bug manifests only when events occur within specific execution contexts C, where C represents conjunctions of: i) application-environment interactions, ii) function execution sequences, or iii) temporal fault orderings.

In other words, an EFIB is triggered by an event E happening when the application is in execution context C.

Rose requires developers to provide the system binaries, a representative workload and a bug oracle. Optionally, developers can also provide a list of functions/files that control critical functionality however, as we will see later (§6), this is not necessary for most bugs. The workload serves to drive the system during testing and the bug oracle is used to identify the presence of a bug.

4.1 Overview and Workflow

To reproduce EFIBs, Rose follows the workflow depicted in Figure 1 which consists of four phases.
Profiling Phase: Before production, Rose identifies common faults, collects system call frequency, and optionally identifies infrequent functions that might represent important system state changes.
Tracing Phase: During production, Rose deploys a production tracer to capture relevant system events. When a bug occurs, as indicated by the bug oracle, it dumps the trace to disk.
Diagnosis Phase: After a bug occurs, Rose analyzes the trace to identify *fault contexts* that might trigger the bug and generates the corresponding fault schedules. Fault contexts are sequences of necessary conditions that must be observed before faults are injected.
Reproduction Phase: Finally, Rose executes the fault schedule using the oracle to check whether the bug is reproduced. If the bug is not found, it passes the information collected

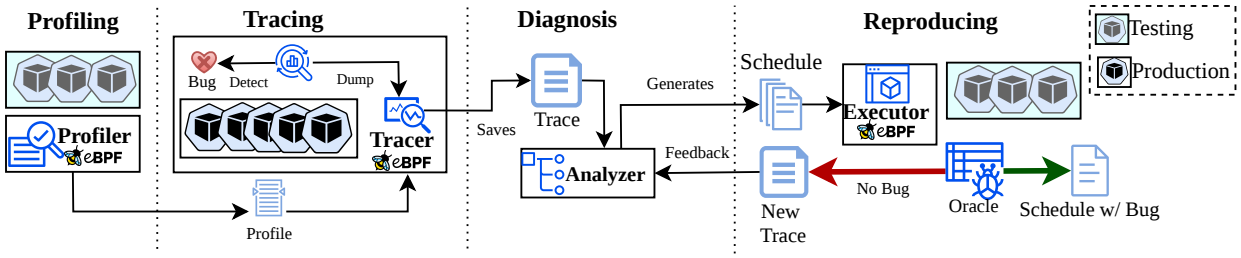

Figure 1. Workflow of Rose.

during execution to the diagnosis phase, which will use it to generate the next fault schedule. Otherwise, it repeats the execution to assess whether this schedule reproduces the bug with a configurable success rate (we used 80% in our evaluation). Rose terminates once a schedule reproduces the bug with the target success rate or if it can not generate more schedules.

4.2 Profiling

Rose can reproduce most bugs (15/20 in our evaluation) using only externally observable behavior. However, for certain bugs, this information is not enough to reproduce the bug with an acceptable rate. This is because some bugs only manifest when the system is executing specific functions.

The goal of this phase is to identify which functions might be relevant through a frequency-based heuristic. Developers provide a list of functions or files that control critical system functionalities (e.g: recovery procedures, leader change mechanisms, snapshotting, etc). Rose automatically extracts the functions addresses/offsets from the binaries and runs the system with the representative workload in a failure-free testing environment. It counts function invocations, separating them into frequent (a configurable value, defaults to 2 calls per second) and infrequent groups. Frequently called functions are discarded, and the rest are passed as monitoring sites (in addition to the system calls, monitored by default) to the tracing phase. The intuition is that EFIBs typically involve code paths that rarely execute during normal operation. This approach provides just enough internal state visibility to capture critical execution contexts while requiring only high-level information from the developers. Rose does static binary analysis to collect the offsets for the application functions, which are used in the Level 3 Diagnosis analysis (Section 4.4.3).

Rose also collects information about any faults that might occur since some faults are common even in failure-free executions, as well as the frequency of system calls, to distinguish between common and rare system calls. This information is used to guide the search in the diagnosis phase.

4.3 Tracing

In this phase, Rose monitors the target system, keeping a sliding window of relevant events (1 million by default), and provides a `dump` primitive to write the window to disk. This primitive is externally invoked either manually by an operator or, more typically, by the existing monitoring infrastructure when a deviation from normal behavior is detected. The tracer runs on each application node. Traces collected from different physical machines are merged in timestamp order. Therefore having loosely synchronized clocks in production allows the diagnosis and reproduction phases to be more effective.

4.3.1 Events. The trace is a sequence of events where each event E_i is associated with a timestamp ts, the *type* of event, and event specific information I. There are four event types.

$$Trace = \Big(E_i = \{\text{ts}, \text{type}, I\}\Big)_{i=1\ldots n}$$

System Call Failures (SCF). The most common event is system call failures (SCF). Even though SCFs are not uncommon, they provide key insights into the (unsuccessful) interactions between the system and the external environment. The tracer monitors system calls and when they return an error, it records the process id, the system call id [31], the file descriptor (for file-related I/O operations), and the `errno` return code. For file-related system calls that use the filename instead of the file descriptors, such as `open` and `stat`, it captures the filename. The filename allows Rose to have a richer context about the possible problematic event, and the `errno` allows to later replay and emulate this event.

$$SCF_I = \{\text{pid}, \text{syscall\_id}, \text{fd}, \text{filename}, \text{errno}\}$$

Application Functions (AF). The tracer records infrequent function invocations which captures important information about application state changes. Rose saves the process id and a function id, a unique integer associated with each function defined during the profiling phase.

$$AF_I = \{\text{pid}, \text{function\_id}\}$$

Network Delays (ND). Network partitions and failures are a common cause of EFIBs. Network failures capture a wide-range of general behaviors such as packet loss, link flapping,

or latency spikes, among others, while network partitions capture specific failure patterns where the system is split into two or more subsets of nodes: nodes are able to communicate with other nodes within the subset but not with nodes in other subsets (either in one or both directions).

While some network failures can be captured as system call failures (e.g.: a `connect` failure) others cannot (e.g.: latency spikes). Temporary network partitions/failures also pose a problem — while, for instance, a failed `connect` might indicate the beginning of a network partition, since Rose does not track successful system calls it is not able to detect when the partition is healed. We therefore need a more robust method of detecting network issues and their duration.

Since periods of network inactivity may indicate partitions or connectivity issues, we employ a strategy that aims to detect network delays as a proxy for identifying possible network failures or partitions. Rose detects these delays by keeping a map of active connections. When a node sends a packet, Rose calculates the delay between this packet and the last packet sent in this connection. If the delay is longer than configurable value (5 seconds by default), Rose records a network delay event, containing the source and destination IPs, the duration, and the number of packets sent in the connection up until this moment.

$$ND_I = \{Dest\_IP, Source\_IP, Duration, Packet\_Count\}$$

Process States and Restarts (PS). Process crashes and pauses are other common source of EFIBs where the crash/-pause of a process triggers a cascading effect in the rest of the system. Rose keeps track of processes state — if a process is in the `waiting` state for more than a configurable time (3 seconds by default), it records this as a possible fault. Rose also creates an event when a process crashes. It records the process id, the state of the process, and for process pauses, the duration of the pause.

$$PS_I = \{\text{pid}, \text{State}, \text{Duration}\}$$

Event Duration. For events that happen over a time interval, namely network partitions or process pauses, Rose records their duration and adds a single event to the window. This minimizes the number of events, but also requires carefully handling events that did not yet terminate when a `dump` is requested. For process pauses, if there are ongoing pauses, Rose saves them to the trace. For network partitions, Rose checks the delay between the last seen packet and the current timestamp and adds it to the trace if it surpasses the configured delay for detecting network partitions. If the tracer is deployed on multiple nodes, the individual traces are merged before being passed to the diagnosis phase.

4.4 Diagnosis

The goal of this phase is to identify the necessary *fault context* in which faults should occur, and output a *fault schedule* that reproduces the EFIB. We define *fault context* as the sequence of necessary conditions that must be observed before a fault is injected. These conditions might include invocation of application-specific functions (AF), particular process states (PS), network delays (ND), and other system call failures (SCF). Note that we do not aim to perform root cause analysis and pinpoint the root of the EFIB, but instead to quickly create a fault schedule that triggers the bug with a high replay rate. This creates tension between precision (capturing the right conditions) and relevance (creating a minimal schedule that consistently triggers the bug).

To this end, we employ an iterative algorithm that starts with a basic context (the faults themselves) and iteratively expands the context of each fault with either the AF events that preceded it, or in the case of SCF, successive invocations of the system call. For every iteration, the algorithm outputs a schedule that is passed to the reproduction phase. If the schedule triggers the bug with a configurable replay rate, Rose stops, otherwise it refines the schedule based on the execution feedback and tries again. The algorithm builds the fault context in three different levels, as depicted in Figure 2.

4.4.1 Level 1: Initial Guess. Rose starts by collecting all the faults in the trace and, since many SCF are benign (i.e. do not trigger a bug), it discards the ones observed in the profiling phase by comparing the syscall name and return code. As an example, upon start a system might invoke `mkdir` to create directories. If they exist, the system call returns `EEXIST` which is expected benign behavior. Given the filtered set of faults, the challenge is how to select which ones to contextualize first, to create a schedule within an acceptable time. Rose prioritizes faults based on their potential impact and frequency. In detail, it prioritizes first PS, then ND, and finally SCF, and within each category it follows chronological order. The rationale is that later faults might be due to earlier faults and hence be a symptom rather than a cause.

The algorithm traverses the sorted list of faults, creating a schedule whose context is only the faults' information (order, inputs for SCF). The key insight is that some bugs do not need a particularly complex application state and can be revealed by simply injecting the faults in the correct order. As an example, to reveal bug `ZOOKEEPER-3006` (Table 1), it suffices to injecting a SCF when reading the snapshot file.

SCF are injected by manipulating the error code, PS by crashing or pausing the process for the relevant time and ND by injecting the respective fault for the duration observed in the trace. Despite its simplicity, this level quickly identifies EFIBs with straightforward trigger conditions and can create a schedule with 100% replay rate for 10/20 bugs (Table 1).

4.4.2 Level 2: What happened before? If Level 1 fails, we expand the fault context with the application states, considering the application-specific functions (Figure 2).

System Calls. To contextualize system calls which have input information (e.g.: the filename), we generate schedules that fail the system call at different invocation counts. As an

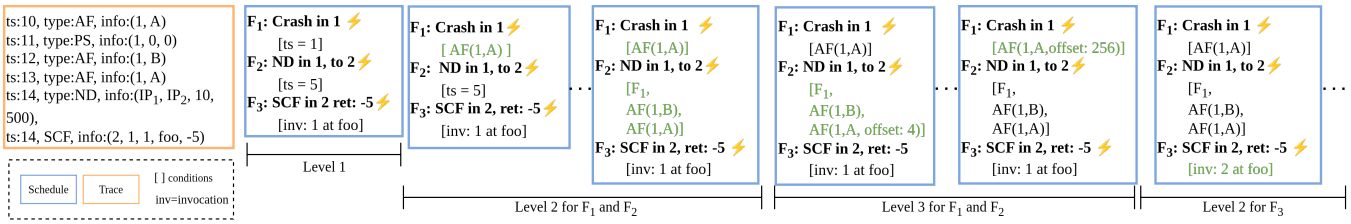

Figure 2. Diagnosis Phase.

example this could be failing the first write to a given file, then in the next schedule the second write, and so on. For system calls without inputs, we attempt to fail the system call the number of times it appeared in the profiling trace, with a hard cap of attempts (50 by default). As an example, bug `HDFS-15032` (Table 1) is triggered 100% of the times by injecting a fault in a specific `connect` invocation.

Process and Network faults. Since these faults do not have context other than a timestamp and duration, Rose uses as context previous AF invocations and earlier faults. This procedure is shown in Algorithm 1 and is based on the insight that faults trigger specific code paths to handle them.

We denote faults with F and functions with `fn`. For a given fault F, observed on node N, the goal is to create the context C of unique functions preceding F. Given the function `fn`, that immediately preceded F on N, we execute a schedule S, where F occurs after `fn` (line 6). Next, we check for the bug with the provided oracle; if it is present, we confirm if S has the target replay rate (line 13). If yes, we return S and terminate, otherwise we continue. If we observe $C \cup$ `fn` functions in the testing run in the same order as production but the bug is not triggered (line 14), it means our sequence is not precise enough to reproduce the bug. Thus we move to the next function (line 7). If we do not observe `fn` in the run, then F is not injected. This means that `fn` is not likely part of a specific code path to trigger the bug, thus we remove `fn` from C and move to the next fault (line 8).

To generate a new schedule, we search for the function `fn'` that immediately precedes `fn`. If `fn'` is not in C, we add it to the schedule and execute, repeating the process, otherwise we ignore it and move to the next fault (line 4). The intuition for ignoring repeated functions is twofold. First, functions that appear more than once are less likely to indicate critical states changes. Second, it reduces the search space without impacting the replay rate (§6).

Role-Specific State. In many systems, different nodes assume different roles such as the primary replica in a primary-backup database deployment. Thus, when a fault occurs, different nodes might show different behaviors contingent on the assumed role. To handle these scenarios, we employ an *Amplification* step that replicates the schedule across all nodes to determine whether the context is role-specific. We do *Amplification* if we did not observe `fn` in the testing run.

Algorithm 1: PS and ND Context Creation

```
▷ S: Schedule, maps faults to context. S[F] denotes the context of fault F
▷ F: Fault we are creating the context for
▷ AF: Ordered set of functions which precede F in the production trace
▷ ufc: unique function counter
▷ run(S) runs schedule S returning the trace and a flag indicating the
  presence of the bug
▷ get_AF(trace,F) gets AF from a trace, which precede F
1  fn findContextforFault(S,F,AF)
2      (C, ufc) ← (∅, 0)
3      for fn ∈ AF do
           ▷ If function is already in context finish and return C
4          if fn ∈ C then return C
5          C ← C ∪ {fn}
6          scheduleResult ← evaluateSchedule(S[F] ∪ C, F, AF, ufc)
7          if scheduleResult then ufc ← ufc+1
8          else return C \ fn
9      return C
10 fn evaluateSchedule(S, F, AF, ufc)
11     (trace, bug) ← run(S)
12     if bug then
           ▷ Success, bug reproduced with target replay rate
13         if confirmBug(S) ≥ 0.8 then exit
14     orderOk,faultInjected ← processTrace(trace,F,AF,ufc+1)
15     return orderOk ∧ faultInjected
16 fn confirmBug(S)
17     (bugRuns, correctRuns) ← (0, 0)
18     for i ∈ {0, 1, 2, ..., 10} do
19         (_ , bug) ← run(S)
20         if bug then bugRuns ← bugRuns+1
21         else correctRuns ← correctRuns+1
           ▷ Abort early if we can not reach the target rate
22         if correctRuns > 2 then return 0
23     return bugRuns/10
24 fn processTrace(trace, F, AFOriginal,size)
       ▷ Get AF from the new trace
25     AFTrace ← get_AF(trace,F)
       ▷ Compare behavior in production and caused by schedule
26     orderOk ← |AFTrace[0..size] ∩ AFOriginal[0..size]| = size
27     faultInjected ← F in trace
28     return (orderOk,faultInjected)
```

If `fn` does not appear in any node, it means the context is not role-specific, and we revert this step, otherwise we keep it in the schedule. We do not employ this for ND, since they have consequences on the entire deployment. As an example, reproducing `RedisRaft-51` (Table 1) is role-specific since it requires injecting the fault when the leader sends the snapshot to other nodes.

Fault Order. To precisely replay the production run, we enforce in testing the fault order that was observed in production as detailed in §4.5.1.

Pruning Runs. During contextualization, bugs might manifest below the target replay rate due to incomplete or imprecise contextualization. To address this variability, when we first detect a bug with fault F, we immediately evaluate its replay rate. If the bug reappears when contextualizing the fault, we save the schedule as a possible candidate, and in the end check the replay rate of all candidates.

4.4.3 Level 3: Function-Specific Context.

In the previous level, and since we capture only the function invocations without any parameters, the faults are injected at the function entry point. Naturally, this misses the fact that functions can contain multiple execution paths, only some of which might be relevant for triggering and reproducing EFIBs. This can be the difference between a failed and a successful reproduction. With this in mind, this level creates more refined schedules that inject faults at specific offsets within the functions that immediately precede each fault. Rather than testing all offsets in sequence, or at random, we prioritize as follows: i) call sites to system calls, ii) call sites to other functions, which can reveal system state changes relevant to the bug, and iii) the rest of the offsets.

For example, reproducing `RedisRaft-NEW` requires crashing a node executing a `write` within a specific function. This corrupts the snapshot and prevents the node from restarting.

4.5 Reproduction

This phase executes the system with the provided representative workload in the testing environment and injects the faults specified in the fault schedule. After each execution, Rose uses the bug oracle to check whether this schedule triggers the bug or not. In practice, the oracle can be implemented by parsing logs to look for specific errors, automated tools such as Elle [32] that check for invariants over the system state, or health checks performed by the production monitoring system, among others. If the bug is found, the execution is repeated with the same schedule to confirm whether it reproduces the bug with the target replay rate. Otherwise, the information collected during the execution is fed back to the diagnosis phase, specifically, the information about injected faults and their contexts (lines 34—35).

The *Executor* drives this phase by: i) tracking the system state, ii) evaluating the state to determine whether a fault should be injected, and iii) injecting the fault.

4.5.1 State Tracking.

Since different nodes are likely to be subject to different faults during a testing run, we start by pre-processing the fault schedule to determine which events apply to which node. Next, we determine for each node the conditions that should be observed in order to inject a fault.

A condition can be a function call or a system call counter (with or without specific inputs), e.g. the number of times that `open` must be called with a specific path. Finally, to preserve the fault order observed in production, we add as conditions to the fault any previous faults that might appear in the trace. This prevents premature or out-of-order fault injection and eliminates a source of randomness, contributing to better replay rates. For example in Figure 2, function A is part of the *fault context* of fault F_2, meaning we must observe A after injecting F_1 in order to inject F_2. Otherwise, if we observed A before injecting $F1$, this would cause us to inject $F2$ before $F1$ violating the fault order observed in production.

4.5.2 Fault Injection.

When all the conditions that satisfy the fault context are observed, the fault is injected. The key concern when injecting faults is ensuring precision, i.e. that faults are injected exactly at the same point in the application state across testing runs. The low-level mechanisms to inject the faults are tied to eBPF's functionalities and are discussed in more detail in the implementation (§5), here we focus only on the high-level design and goals. To fail system calls, we override the return value with the one described in the fault schedule, and completely skip the logic of the system call. This emulates a scenario where the system call failed right at the beginning of the invocation. Since we can not know from the production trace whether the system call completed, we opt to completely skip their execution and return the error code observed in production. This decision is based on the fact that skipping a system call is a faulty behavior and by returning an error, we force the system to go into the code paths that handle faults. For process pauses and crashes, we send a signal from kernel space to the target process which ensures the process is crashed/paused consistently in the same state. For network faults, we drop packets between the target processes as specified in the schedule.

5 Implementation

Rose is written in C and Rust for the performance-critical components, and Python for the utilities and orchestration between the different steps of the workflow (Figure 1).

5.1 Profiler

The Profiler identifies infrequent application functions with static and dynamic analysis, and collects information about the frequency of system calls. It receives a binary and a list of source file names. For static analysis, we use a Python script that employs standard Linux utilities (`readelf` and `addr2line`) to extract function symbols and their corresponding binary offsets from the target executables. We then collect all of the symbols defined in the provided files. For dynamic analysis, we leverage an extended version of the production tracer that also counts functions and system calls invocation frequency. The Profiler outputs a file containing function names, binary offsets, and invocation frequency statistics, which guides subsequent tracing operations.

5.2 Tracer

The tracer is performance-critical since it runs alongside production systems. It is implemented using `libbpf-rs` [33], a Rust binding for the Linux eBPF C library. We selected Rust for its memory safety guarantees and performance comparable to C/C++ implementations. The tracer consists of eBPF programs written in C (as required by the eBPF infrastructure) with a Rust control plane that manages program loading, event collection, and trace persistence.

We maintain a fixed-size circular buffer of recent events (1 million by default) using a `BPF_MAP_ARRAY`. This provides a bounded memory footprint and eliminates continuous disk I/O since the buffer is only dumped to disk when triggered by the bug oracle, minimizing the impact in production.

Instead of capturing all system calls, we focus exclusively on failures, which are relatively rare but contain the most valuable information about potential faults. To achieve this, we leverage eBPF `sys_exit` tracepoint, which is called every time a system call returns. We only keep invocations that returned an error and discard all the others. Similarly, we trace only infrequent application functions (identified by the Profiler) rather than all function calls, significantly reducing the number of traced events. Function calls are traced through eBPF user-function probes.

File-Based System Calls. For file based system calls, it is important to save the full path they operate on. Instead of saving the full path for every system call (which would require expensive string copies), we maintain a lightweight mapping of filenames to file descriptors, which is only updated during `open`, `close`, and `dup` operations. To minimize runtime overhead, we reconstruct the path information for each system call in a post-processing step outside the critical path, after saving the trace to disk.

For system calls that operate directly on path names rather than file descriptors (e.g. `open`, `stat`), we record only the userspace address of the variable containing the path name in the `sys_enter` tracepoint. If the system call fails, we collect its contents. This way we avoid unnecessary copy operations.

Network Delays. For network delay detection, we leverage XDP (eXpress Data Path) programs attached directly to network interfaces, enabling packet analysis at the earliest point in the networking stack with near-zero overhead. XDP only supports attaching to the network ingress, thus it only detects packets on the receiver side. This means that ND detection is not affected by sender side packet retransmissions.

Process State. Rose uses `procfs` [34] to query the process state at regular intervals (default of 1 second). These implementation choices enable the tracer to maintain overhead below 2.6% per node (see §6) while capturing sufficient information for fault reproduction.

5.3 Analyzer

The analyzer implements the diagnosis phase and receives the profile, a buggy production trace, the workload, oracle, and the application binary. It parses this information and generates a C program implementing the schedule.

The program is passed to the executor which runs the schedule, uses the oracle to check for the presence of the bug, and collects the trace which is used to refine the schedule.

5.4 Executor

The executor is implemented in C and uses the `libbpf` [35] framework to load, verify, and attach the eBPF programs. These eBPF programs are system call probes, kernel probes, and user function probes.

To emulate system call faults, we use the kernel system call probe `bpf_override_return` which allows overriding the return value of the probed function. We selected kernel probes instead of `tracepoints` because they enable the use of `bpf_override_return` while providing the same level of contextual information (timestamps, input arguments, etc.).

To emulate network faults we use custom *Linux TC* filters. This is achieved by inspecting the packet headers, checking the source and destination IPs, and dropping the packets according to the details of the fault.

Process faults are injected with the `bpf_send_signal` eBPF helper function. This helper enables sending signals to a process from kernelspace. We use this approach instead of relying on userspace signals to ensure that the process state manipulation occurs at precisely the intended execution point before the process leaves kernel mode.

Function Offset Fault Injection. To inject faults in specific function offsets, we insert `uprobes` in the respective binary addresses. When execution reaches these addresses, the `uprobes` are triggered and Rose injects the fault.

Tracking Process Ids. As discussed in §4.5, state tracking and fault injection are done on a per-process basis. Rose uses eBPF maps to map the process id to the faults and the respective conditions the node (a specific process id) must reach to inject the fault. Since systems commonly spawn child processes to perform certain tasks, there might be scenarios where it is a child process that reaches the necessary conditions for fault injection. To solve this issue, we keep a map of child processes to the parent process in the schedule execution. The decisions are made according to the initial process id, and the faults are injected in the parent. When injecting a process crash, if the node restarts it will have a different process id. To keep the mapping consistent, we assign the new process id to the original one to make decisions, and inject faults on the new process id.

6 Evaluation

The evaluation aims to answer three main questions: i) how effective is Rose in automatically reproducing EFIBs, ii) what

is the tracer's overhead, and iii) what is the impact of the heuristics. All experiments were done on a machine with 2x Intel(R) Xeon(R) Gold 5320 CPU @ 2.2GHz (52 cores) with 64GB of RAM running Linux 6.11.1.

6.1 System Selection and Methodology

To select the systems and the EFIBs to reproduce, we used Jepsen [10] and Anduril [18] reports, and a preliminary manual bug search.

Jepsen [10] is a randomized fault-injection tool that maintains a public database of bugs. We studied Jepsen's analysis of the last 4 years using bug quantity and diversity as selection criteria for EFIBs. In the end, we selected RedisRaft [30, 36] and Redpanda [37, 38].

RedisRaft is an extension of the Redis in-memory key-value store, that aims to offer strict serializability through the Raft consensus algorithm. From the bugs found by Jepsen, we ruled out bugs that did not depend on external events, for example, manual membership changes, since these are not EFIBs. We also ignored bugs that required Elle [32] a transactional consistency checker databases, which Jepsen uses as a bug oracle. This main reason to ignore these bugs as due to the long execution time of Elle which needs to analyze the entire transaction history.

From a total 21 bugs, we eliminated 13 which where not EFIBs, and 3 that required Elle as an oracle. Out of the 5 bugs, we managed to obtain traces that reproduced the bug for 4 of them and were able to automatically reproduce 3. The bug that Rose did not automatically reproduce is explained by the dependency of another bug also happening [36]. `RedisRaft-NEW` and `RedisRaft-NEW2` were two unreported bugs we uncovered when testing with RedisRaft.

Redpanda is a high performance, Kafka-compatible streaming data platform designed to handle real-time data processing. Similarly to the RedisRaft methodology we discard bugs which are not EFIBs however for this system we considered bugs that require Elle to illustrate that Rose can be used with complex oracles. Jepsen reported 11 bugs (note that bug #1 and #3039 in the report are the same), 6 of which are not EFIBs. Out of the 5 bugs, we managed to obtain traces that trigger the bug for 3 of them, 2 of which we reproduce automatically. These bugs needed Elle to be revealed, and always occur together due to having the same source defect. For the bug Rose did not reproduce automatically, this stems from the fact that we were unable to compile the binary with debug information due to outdated/missing dependencies, and hence Rose could not run the Level 3 analysis.

Anduril [18] mimics the occurrence of faults by injecting exceptions in Java-based programs. Anduril reports 22 bugs, we encountered problems either running or compiling 8 of them, including all Cassandra ones[1], which leaves us with 14 bugs. One of these cannot be reproduced by Rose since it depends on a `Interrupted Exception` which is an internal Java exception rather than an EFIB. We reproduced 10 out of 13 remaining, targeting system diversity – the remaining three are HBase-specific and are all due to `IOException`.

Developer Inputs. We obtain the traces, which would come from a production environment by running each system with our tracer. For the bugs reported by Anduril we rely on the provided unit tests or small cluster deployments which serve both as a workload and a oracle since the test outcome indicates whether a bug is present. For the bugs reported by Jepsen, we run the system with the workload and faults provided by Jepsen. In terms of oracles, for Redpanda we directly used Elle [32]. For RedisRaft, we check for specific messages in the logs, following the details provided in Jepsen's analysis.

To identify the relevant files, we performed a high-level manual inspection of the code and searched for patterns (e.g.: with `grep`) indicating relevant functionality (e.g.: snapshotting, partitiong, etc). Even as researchers unfamiliar with the respective code bases this proved to be a simple task.

6.2 Bug Reproduction

Table 1 shows the EFIBs automatically reproduced by Rose. Column *Faults Inj.* reports the faults that had to be injected to uncover the bugs, *RR (%)* reports the replay rate, *Sched.* reports the number of schedules generated before achieving the target replay rate, *#R* reports the total number of runs, *Time (m)* reports the total time taken, and *FR* reports the percentage of faults that are discarded by comparing the buggy trace with a normal execution. Recall that after finding a schedule that reproduces the bug, we execute the same schedule ten more times to measure the replay rate. Hence, in most cases, the number of runs is the number of schedules generated plus ten. When Rose can not find all the fault context, there is some variance in which case we repeat the process 3 times and show the average and standard deviation.

Overall, the average time to reproduce an EFIB with the target replay rate is 60 minutes with a standard deviation of 90, mainly due to `Redpanda-3003` and `Redpanda-3039`. These two bugs have an average replay rate below the target (80%) leading to longer test times. Besides these bugs require Elle which takes about about 2 minutes to analyze the transaction history of each fault schedule.

In summary, the results show that Rose is quick at automatically deducing the schedules to reproduce EFIBs. Due to space constraints we omit a detailed discussion of all the bugs and discuss just a few representative ones.

Case Study: `RedisRaft-43`. To illustrate Rose diagnostic capabilities, we examine the `RedisRaft-43` bug (our motivating example in §3), where Jepsen's random fault injection achieves only a 1% replay rate. The system has 5 nodes.

Rose starts with the trace containing the bug and systematically refines the schedule until the target replay rate is achieved. The *Level 1* analysis (§4.4.1), considers only basic

[1]We contacted the authors but did not obtain a reply

Bug	Src.	Description	Faults Inj.	RR(%)	Sched.	#R	T(m)	FR(%)
RedisRaft-42 [39]	J	Node crashes due to failed assert related to snapshot & log integrity.	PS(Crash)	100	1	11	22	37.5
RedisRaft-43 [29]	J	Snapshot index mismatch.	PS(Crash)*3+ ND + PS(Crash)	100	19	29	58	7
RedisRaft-51 [40]	J	Node crashes due to failed assert related to cache index integrity.	PS(Pause)*3	90±8	10±1	28±4	56±7	7
RedisRaft-NEW [41]	J	Redis itself crashes due to an inconsistent snapshot file.	ND + PS(Crash) + PS(Crash)	100	22	32	70	22
RedisRaft-NEW2 [2]	J	Redis itself fails due to a repeated key.	ND	100	1	11	11	41
Redpanda-3003 [42]	J	Redpanda fails to perform deduplication of sent messages	5*PS(Pause)	70±14	12±1	81±20	324±82	56
Redpanda-3039 [43]	J	Inconsistent Offsets.	5*PS(Pause)	70±14	12±1	81±20	324±82	56
Zookeeper-2247 [44]	A	Service becomes unavailable when leader fails to write transaction log.	SCF(write)	100	5	15	15	86
Zookeeper-3006 [45]	A	Invalid disk file content causes null pointer exception.	SCF(read)	100	1	11	5	68
Zookeeper-3157 [46]	A	Connection loss causes the client to fail.	SCF(read)	100	1	11	20	84
Zookeeper-4203 [47]	A	The leader election is stuck forever due to connection error.	SCF(accept)	73±16	16±3	34±12	34±12	92
HDFS-4233 [48]	A	NN keeps serving even after no journals started while rolling edit.	SCF(openat)	100	1	11	11	95
HDFS-12070 [49]	A	Files remain open indefinitely if block recovery fails.	SCF(fstat)	100	20	30	77	96
HDFS-15032 [50]	A	Balancer crashes when it fails to contact an unavailable namenode.	SCF(connect)	100	26	36	57	96
HDFS-16332 [51]	A	Missing handling of expired block token causes slow read.	SCF(read)	100	1	11	14	96
Kafka-12508 [52]	A	Emit-on-change tables may lose updates on error or restart.	SCF(openat)	100	1	11	22	92
HBASE-19608 [53]	A	Race in MasterRpcServices.getProcedureResult.	SCF(openat)	100	1	11	11	96
MongoDB:2.4.3 [54]	M	MongoDB Data Loss Jepsen report.	2*ND	100	1	11	22	16
MongoDB:3.2.10 [55]	M	MongoDB Unavailability Jepsen report.	ND	100	1	11	22	50
Tendermint-5839 [56]	M	Does not validate permissions to access file.	SCF(openat)	100	1	11	5	80

Table 1. Bugs reproduced by Rose, J=Jepsen, A=Anduril, M=Manual search, Inj=Faults Injected, R. R=Replay Rate, Sched=Number of Generated Schedules, R#=Number of runs, T=Total Time (min), FR=Faults Removed.

fault order and produces a schedule that: i) crashes three non-leader nodes, ii) creates a network partition that isolates the leader, and iii) crashes the leader.

This schedule fails to consistently reproduce the bug and hence Rose moves to *Level 2* analysis (§4.4.2) which expands the fault's context. Rose incrementally adds functions to the context (whose invocation must be observed before the fault is injected), eventually identifying that the fault must occur when the `RaftLogCreate` function is executing, significantly narrowing the fault injection window. This function immediately calls `prepareLog`, however, since in the Level 2 Rose injects the fault (process crash in this case) right at the beginning of the invocation, `parseLog` is never run.

The specificity at which exact point in the execution the fault is injected is critical since the bug manifests only when the process is crashed before the invocation of `parseLog`, and explains the very low replay rate of a randomized approach. With this precisely defined fault context, Rose reliably achieves a 100% replay rate across multiple test iterations. Upon restarting, the node fails the assertion that the indexes between the log and snapshot must match, due to the fault injected by Rose. This was fixed in commit `d1d728d` [57] which changed the `RaftLogOpen` function to not rebuild the index but instead keep the one in the log.

Case Study: `RedisRaft-NEW`. This bug is unreported by Jepsen but appeared in one of the traces we collected when trying to produce a trace for `RedisRaft-43`. This new bug presents an instructive case for the precision of fault injection since the last fault fault must occur at a specific point in execution leading to a corrupted snapshot.

The *Level 1* analysis produces a schedule that: i) creates a network partition that isolates the leader, ii) crashes the original leader, and iii) crashes the original leader again after it restarts. This schedule does not trigger the bug, and hence Rose moves to *Level 2*. Here the last fault differs from the previous bug, with the last process crashed being contingent on the invocation of function `storeSnapshotData`. Still, this is not enough to reproduce the bug, and hence Rose moves to *Level 3* (§4.4.3) and starts exploring the function offsets. Rose prioritizes invocations to system calls and in the `storeSnapshotData` there are three: an `open`, a `write`, and a `close`. Rose generates schedules that injects the faults at each of these points, and when the fault happens at the `write` invocation the bug is triggered — upon restarting the snapshot is corrupted due to a mismanagement of the snapshot file. This bug requires very specific conditions which is why, we think, it was not reported by Jepsen and once again illustrates the importance of precise fault injection — the bug is only triggered if the system crashes within a specific function and when executing a specific system call.

Case Study: `Zookeeper-3006`. In this bug, a node attempts to calculate the snapshot size. If an exception occurs while accessing the snapshot (due to a failed `read`), the exception is correctly caught but the value holding the snapshot size is still used internally, causing a `NullPointerException` that crashes the node.

By comparing a faultless trace and the provided one, Rose quickly found the `read` system call that caused the bug. Next, in the Level 1 analyses, Rose took an initial guess, by failing the first read on file `snapshot.0`. This read is indeed used to calculate the snapshot size, and failing it triggered the bug

in the first attempt. Rose then runs the schedule 10 times to assess the replay rate, which in this case is 100%.

6.3 Tracer Overhead

Since the tracer is designed to run in production, the primary metric for production viability is overhead. We performed a comparative study using three tracing approaches: Rose tracer as described in §4.3 and §5.2, a *Full* approach that records all the system call invocations, and an *IO content* approach that captures the same events as the Rose tracer plus the contents (up to 128 bytes) of every `read` and `write`.

We used a 3-node Redis cluster under YCSB [58] workload A (50% reads, 50% updates) and measured the throughput degradation compared to a baseline without tracing. The results are shown in Table 2. Column *Events* reports the events captured by the tracer, *Saved* reports the number of events kept in the circular buffer, *Memory* reports the maximum memory usage, *Time* reports the processing time of the trace, and *Overhead* reports the application level overhead.

By capturing only the essential events (failed system calls) Rose has to process substantially fewer events (5,444 vs. 1,048,576), which results in a minimal memory footprint (712 KB vs 151 MB) and enables rapid processing (0.06s vs. 17s). In terms of the application overhead, Rose's tracer has an 2.6% overhead per node, while *Full* has an 3.9% overhead showing the benefits of only recording system calls that fail. As expected, recording more information as done by *IO Content* increases the overhead to 4.9% mostly due to the cost of memory copies. The memory footprint also increases due to saving the contents of write and read operations.

These results show that Rose selective tracing successfully minimizes performance impact while capturing all essential information to reproduce EFIBs.

Table 2. Cost of Rose tracer versus other alternatives.

Approach	Events	Saved	Memory	Time (s)	Overhead
Rose	5,444	5,444	712 KB	0.06	2.6%
Full	14M	1,048,576	151 MB	17	3.9%
IO Content	9M	1,048,576	281 MB	17	4.9%

6.4 Heuristic Effectiveness

Table 3. Effectiveness of the function frequency heuristic.

Bug	All Func.	Infreq. Func. (default)	Reduction %
`RedisRaft-43`	1,699,348	3,677	99.7
`RedisRaft-51`	214,552	2,121	99.0
`RedisRaft-NEW`	3,023,112	4,895	99.8
`Redpanda-3003/3039`	1,749,429	11,842	99.3

We now evaluate the effectiveness of Rose heuristics for discarding frequent function calls and faults that have been observed during normal execution. Regarding the first heuristic, we did an experiment with the heuristic disabled which resulted in Rose tracing all the functions from the user provided files. Results are shown in Table 3 for the scenarios where this heuristic was active and show that it reduces the number of traced functions by up to 4 orders of magnitude. The heuristic enables tracing to be lightweight enough to be used in production environments as otherwise tracing overhead would be too high, due to frequent context switches [60]. Furthermore, since the ring buffer only stores the last 1 million events in the ring buffer, the trace would be polluted with irrelevant calls (in 4 out of 5 affected bugs the number of functions is larger than the buffer), reducing Rose effectiveness in reproducing EFIBs.

The heuristic that discards failed systems calls that were observed in a normal execution is also very effective in reducing the search space, by up to 96% (column *FR* of Table 1). For example, in a failure-free execution of `ZOOKEEPER-2247` we observed that `stat` and `readlink` failed 312 and 614 with `ENOENT` and `EINVAL`, respectively, which are expected and benign failures related to checking for the existence of files.

The rationale behind both heuristics is that it is possible to substantially reduce the search space by getting rid of common events (application functions or failed system calls) without negatively impacting the replay rate since these events represent common code paths that are typically not exercised in the presence of faults.

Finally, we disabled the early halt search when a function is already in the context (§4.4.2). The results show that the replay rate did not improve (hence Rose is not overlooking essential context) and only the number of runs was negatively affected, for instance in `RedisRaft-51` the number of runs increased from 28±4 to 30±9.

6.5 Discussion

The effectiveness of each level in reproducing bugs provides empirical insights into the relative complexity of EFIBs.

The *Level 1* analysis, which uses only information about the faults themselves was sufficient to reproduce 10 of the 20 bugs in our evaluation set. Within this group, 6 bugs required only information about the relative fault ordering, while the remaining 4 required specific information about the system call inputs. These findings suggest that for approximately half of external-fault-induced bugs, basic external observability provides sufficient context for reliable reproduction.

Level 2 reproduced 9 additional bugs. Among these, 7 required identification of a specific iteration of a system call (the *nth* invocation) with specific arguments, while the other 2 manifested only when particular function execution sequences preceded the injection of the fault. The context refinement substantially increases the replay rate with most bugs achieving 90%. The results indicate that while basic context suffices for many cases, function-level execution awareness significantly extends reproduction capabilities.

[2]This did not show after commit 2d1cf30 [59], thus we did not report it.

Finally, only a single bug required the *Level 3* precision of injecting faults at a specific offset within a specific function.

The empirical distribution of bugs reproduced across these three levels validates Rose stratified approach. On the one hand most of the bugs can be reproduced with simpler and faster strategies that only require external observability, validating our main insight that external observability plays a key role in reproducing external-fault-induced bugs. On the other hand, some bugs require specific context about the system state which, while more expensive, can still be obtained in an application-agnostic manner.

7 Related Work

In this section, we analyze existing approaches that use fault injection techniques for bug finding and reproduction.

Bug Finding with Fault Injection. Fault injection systems fall into two categories: general-purpose frameworks and specialized tools. General frameworks like Jepsen [10] use randomized fault injection to test system invariants, while Mallory [11] extends Jepsen with reinforcement learning techniques that reward revealing new behaviors. FaultSee [61] proposes a domain-specific language to specify fault injection scenarios, and Chaos Monkey [62] focuses on process termination within production environments.

Specialized tools target specific system categories or fault types. PACE [12] explores correlated crash vulnerabilities in distributed filesystems by systematically generating persistent states, while Sieve [14] assesses the reliability of cluster management controllers. FCatch [15] aims to find timing-sensitive bugs in cloud systems, while CrashMonkey [16] focuses on uncovering crash consistency issues in file systems. Legolas [63] uses static analysis to extract abstract states from system code to expose partial failure bugs in distributed systems. Several BFT testing frameworks [64–68] use system internals to identify critical fault injection points.

These approaches use various techniques to search the fault space and discover potential bugs. However, their goals are fundamentally different than Rose's, which focuses on quickly and precisely reproducing known bugs. Furthermore, most of these approaches require extensive instrumentation and/or application-level knowledge and hence operate at a higher abstraction level than Rose OS-level approach.

Bug Reproduction. Record-and-replay systems [19–22] capture non-deterministic behavior (e.g., program inputs and thread interleavings) and later replay the exact sequence of events. While comprehensive, they have prohibitive overhead (e.g., RR [22] has a minimum overhead of 1.49×), making them impractical for fault reproduction in production.

ReproLite [69] proposes a domain-specific language to express bug scenarios (similar to our fault schedule) and enforce their execution order. It does not search for fault context to inject faults, but rather serves to validate developers' conjectures about faulty scenarios. Pensive [17] processes system's bytecode and logs to automatically reconstruct the sequence of inputs that trigger a given bug. It focuses on uncovering faulty inputs that trigger bugs rather than bugs triggered by external faults. Anduril [18] aims to reproduce fault-induced failures and is the closest work to ours. It extends Pensieve's algorithm with exception handling, searching for failure-inducing exceptions rather than input sequences.

These tools rely JVM-specific information for application monitoring and fault injection, and hence cannot be broadly applied to non-JVM applications. Rose employs OS-level observation and fault injection techniques and hence can target systems implemented in any programming language. The low overhead of the tracer allows it to run alongside the production system and capture the minimal amount of information necessary to reproduce the bug. Finally, Rose's progressive fault context refinement efficiently identifies minimal reproduction conditions to trigger the bug with high replay rates (>90% in most cases).

8 Limitations and Future Work

We now discuss some limitations of Rose and how these can be addressed in future work.

False Negatives. The current implementation might have false negatives during fault schedule evaluation. When a schedule fails to trigger the bug during the initial testing, Rose discards it entirely, even though subsequent executions might have achieved a statistically significant replay rate. A possible enhancement would involve executing each candidate schedule multiple times to establish statistical confidence, though this linearly increases reproduction time.

Concurrency. The current implementation also lacks support for thread-specific fault contexts. While this information can be captured by the tracer, our fault context refinement does not consider thread-specific conditions. Even though we did not encounter bugs that required this thread-specific context, we plan to add this functionality in future work.

JVM Function Tracing. Rose traces application functions by attaching uprobres to binary addresses which cannot be directly used in JVM-based applications. This limitation can be overcome with further engineering work by leveraging eBPF's `USDTs` [70], or annotations with AspectJ [71].

Unsupported Operations and Bugs. A more fundamental constraint of Rose's design is the ability to handle faults in memory-mapped I/O operations. Since most accesses to memory-mapped files bypass system calls, this creates a blind spot in Rose observability model where faults occurring within memory-mapped regions cannot be detected through system call monitoring alone. Therefore, bugs triggered by memory access errors remain outside Rose current reproduction capabilities. Addressing this limitation requires further research.

EFIBs arguably affect mainly control aspects of the system since they activate code paths that handle erroneous interactions between the system and the environment. Indeed, all the bugs Rose reproduced are related to the control-path which also aligns with Anduril's observations [18]. However, it is possible for EFIBs to affect the data-path, or be data dependent. While conceptually Rose can capture and manipulate the inputs, this is costly to run in production in the current implementation (§6.3) as confirmed by systems that rely on record-and-replay techniques (§7).

9 Conclusion

Reproducing EFIBs is challenging since they are only exposed when external events happen at specific execution points. Existing state-of-the-art tools that target this problem are JVM-specific and emulate faults by throwing exceptions at strategic points in the code. They are limited to the JVM ecosystem and also fail to capture external events that do not necessarily raise exceptions such as network delays.

This paper introduces Rose, a novel approach for reproducing EFIBs by relying only on OS-level monitoring and fault injection mechanisms. Rose is language-agnostic, enabling the reproduction of bugs across diverse programming environments — our evaluation reproduces 20 bugs across eight widely-used distributed systems implemented various languages, namely HBase (Java), HDFS (Java), Kafka (Java/Scala), MongoDB (C++), RedisRaft (C), Redpanda (C++), Tendermint (Go), and Zookeeper (Java). Moreover, Rose tracer captures sufficient information for reliable bug reproduction with a low overhead of 2.6% per node. This allows deployment alongside production systems where performance constraints preclude extensive monitoring, and hence enables capturing information about bugs that happen seldom. Finally, by employing a Diagnosis phase where we progressively refine the context where faults should be injected, Rose builds schedules that reproduce EFIBs with high replay rates (>90%). These characteristics allow developers to be more efficient in reproducing EFIBs and, we argue, contribute to improving distributed systems' reliability.

10 Acknowledgments

We thank our shepherd, Yanyan Jiang, and the anonymous reviewers for their feedback and help in improving the paper. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (https://doi.org/10.54499/UID/50021/2025), UID/PRR/50021/2025 (https://doi.org/10.54499/UID/PRR/50021/2025) and the bilateral project PLD2.

References

[1] "Google lost $1.7m in ad revenue during youtube outage," accessed: 2025-1-01. [Online]. Available: https://www.foxbusiness.com/technology/google-lost-ad-revenue-during-youtube-outage-expert

[2] "Incident report: Spotify outage on april 16, 2025," accessed: 2025-5-13. [Online]. Available: https://engineering.atspotify.com/2025/05/incident-report-spotify-outage-on-april-16-2025

[3] "Incident report: High error rates for chatgpt, apis, and sora, 2025," accessed: 2025-5-13. [Online]. Available: https://status.openai.com/incidents/6bwlxnvdncnm

[4] F. Hackett, J. Rowe, and M. A. Kuppe, "Understanding inconsistency in azure cosmos db with tla+," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2023, pp. 1–12.

[5] A. Avizienis, J.-C. Laprie, B. Randell *et al.*, "Fundamental concepts of dependability," *Technical Report Series-University of Newcastle upon Tyne Computing Science*, 2001.

[6] X. Wang, S. Duan, J. Clavin, and H. Zhang, "Bft in blockchains: From protocols to use cases," *ACM Comput. Surv.*, vol. 54, no. 10s, Sep. 2022. [Online]. Available: https://doi.org/10.1145/3503042

[7] "Google dashoard bug incident," accessed:2025-1-01. [Online]. Available: https://status.cloud.google.com/incident/zall/20013

[8] S. Ghosh, M. Shetty, C. Bansal, and S. Nath, "How to fight production incidents? an empirical study on a large-scale cloud service," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 126–141. [Online]. Available: https://doi.org/10.1145/3542929.3563482

[9] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 155–162. [Online]. Available: https://doi.org/10.1145/3317550.3321438

[10] "Jepsen," accessed: 2023-2-1. [Online]. Available: https://github.com/jepsen-io/jepsen

[11] R. Meng, G. Pîrlea, A. Roychoudhury, and I. Sergey, "Greybox fuzzing of distributed systems," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1615–1629. [Online]. Available: https://doi.org/10.1145/3576915.3623097

[12] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Correlated crash vulnerabilities," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 151–167. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/alagappan

[13] Y. Chen, F. Ma, Y. Zhou, M. Gu, Q. Liao, and Y. Jiang, "Chronos: Finding timeout bugs in practical distributed systems by deep-priority fuzzing with transient delay," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1939–1955.

[14] X. Sun, W. Luo, J. T. Gu, A. Ganesan, R. Alagappan, M. Gasch, L. Suresh, and T. Xu, "Automatic reliability testing for cluster management controllers," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 143–159. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/sun

[15] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "Fcatch: Automatically detecting time-of-fault bugs in cloud systems," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 419–431. [Online]. Available: https://doi.org/10.1145/3173162.3177161

[16] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. Carlsbad, CA: USENIX Association, 2018.

[17] Y. Zhang, S. Makarov, X. Ren, D. Lion, and D. Yuan, "Pensieve: Non-intrusive failure reproduction for distributed systems using the

event chaining approach," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 19–33. [Online]. Available: https://doi.org/10.1145/3132747.3132768

[18] J. Pan, H. Wu, T. Leesatapornwongsa, S. Nath, and P. Huang, "Efficient reproduction of fault-induced failures in distributed systems with feedback-driven fault injection," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 46–62. [Online]. Available: https://doi.org/10.1145/3694715.3695979

[19] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: an application-level kernel for record and replay," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 193–208.

[20] H. Liu, S. Silvestro, W. Wang, C. Tian, and T. Liu, "ireplayer: in-situ and identical record-and-replay for multithreaded applications," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 344–358. [Online]. Available: https://doi.org/10.1145/3192366.3192380

[21] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, "Doubleplay: parallelizing sequential logging and replay," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 15–26. [Online]. Available: https://doi.org/10.1145/1950365.1950370

[22] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 377–389. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan

[23] "ebpf." accessed: 2025-05-15. [Online]. Available: https://ebpf.io/

[24] "bpf-helpers," accessed: 2025-04-29. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html

[25] "bpf-helpers," accessed: 2025-04-29. [Online]. Available: https://docs.ebpf.io/linux/helper-function/bpf_send_signal/

[26] "ebpf tracepoints." accessed: 2025-01-01. [Online]. Available: https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_TRACEPOINT/

[27] "ebpf kernel syscall probe." accessed: 2025-01-01. [Online]. Available: https://docs.ebpf.io/ebpf-library/libbpf/ebpf/BPF_KSYSCALL/

[28] "ebpf user function probes." accessed: 2025-01-01. [Online]. Available: https://docs.kernel.org/trace/uprobetracer.html

[29] "Redisraft bug 43." accessed: 2025-01-01. [Online]. Available: https://github.com/RedisLabs/redisraft/issues/43/

[30] "Redisraft repository." accessed: 2025-05-15. [Online]. Available: https://github.com/RedisLabs/redisraft

[31] "System call identifiers." accessed: 2025-5-11. [Online]. Available: https://github.com/torvalds/linux/blob/v6.7/arch/x86/entry/syscalls/syscall_64.tbl

[32] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *CoRR*, vol. abs/2003.10554, 2020. [Online]. Available: https://arxiv.org/abs/2003.10554

[33] "libbpf-rs - idiomatic rust wrapper around libbpf." accessed: 2025-01-01. [Online]. Available: https://github.com/libbpf/libbpf-rs

[34] "procfs - interface to the proc pseudo-filesystem on linux," accessed: 2025-01-01. [Online]. Available: https://crates.io/crates/procfs

[35] "Documentation for libbpf, a userspace library for loading and interacting with bpf programs." accessed: 2025-1-01. [Online]. Available: https://libbpf.readthedocs.io/en/latest/index.html

[36] "Jepsen analysis of redisraft." accessed: 2025-05-15. [Online]. Available: https://jepsen.io/analyses/redis-raft-1b3fbf6

[37] "Redpanda, a kafka® compatible streaming data platform." accessed: 2025-1-01. [Online]. Available: https://www.redpanda.com/

[38] "Jepsen analysis of redpanda." accessed: 2025-05-05. [Online]. Available: https://jepsen.io/analyses/redpanda-21.10.1

[39] "Redisraft issue 42." accessed: 2025-05-15. [Online]. Available: https://github.com/RedisLabs/redisraft/issues/42

[40] "Redisraft issue 51." accessed: 2025-05-15. [Online]. Available: https://github.com/RedisLabs/redisraft/issues/51

[41] "Redisraft issue 654." accessed: 2025-05-15. [Online]. Available: https://github.com/RedisLabs/redisraft/issues/654

[42] "Redpanda issue 3003." accessed: 2025-05-15. [Online]. Available: https://github.com/redpanda-data/redpanda/pull/3003

[43] "Redpanda issue 3039." accessed: 2025-05-15. [Online]. Available: https://github.com/redpanda-data/redpanda/pull/3039

[44] "Zookeeper issue 2207." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-2247

[45] "Zookeeper issue 3006." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-3006

[46] "Zookeeper issue 3157." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-3157

[47] "Zookeeper issue 4203." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-4203

[48] "Hdfs issue 4233." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/hdfs-4233

[49] "Hdfs issue 12070." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/hdfs-12070

[50] "Hdfs issue 15032." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/hdfs-15032

[51] "Hdfs issue 16332." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/hdfs-16332

[52] "Kafka issue 12508." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/KAFKA-12508

[53] "Hbase issue 19608." accessed: 2025-05-15. [Online]. Available: https://issues.apache.org/jira/browse/hdfs-19608

[54] "Jepsen mongodb:2.4.3," accessed: 2025-05-14. [Online]. Available: https://aphyr.com/posts/284-call-me-maybe-mongodb

[55] "Mongodb unavailability bug," accessed: 2025-05-14. [Online]. Available: https://jira.mongodb.org/browse/SERVER-27125

[56] "Tendermint issue 5839." accessed: 2025-05-15. [Online]. Available: https://github.com/tendermint/tendermint/issues/5839

[57] "Redisraft commit d1d728d." accessed: 2025-05-15. [Online]. Available: https://github.com/RedisLabs/redisraft/commit/d1d728d

[58] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *SoCC*. ACM, 2010, pp. 143–154.

[59] "Redisraft commit 2d1cf30." accessed: 2025-05-15. [Online]. Available: https://github.com/RedisLabs/redisraft/commit/2d1cf30

[60] Y. Zheng, T. Yu, Y. Yang, Y. Hu, X. Lai, D. Williams, and A. Quinn, "Extending applications safely and efficiently," in *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025, pp. 557–574.

[61] M. Amaral, M. L. Pardal, H. Mercier, and M. Matos, "Faultsee: Reproducible fault injection in distributed systems," in *2020 16th European Dependable Computing Conference (EDCC)*, 2020, pp. 25–32.

[62] "Chaos monkey github," accessed: 2025-1-01. [Online]. Available: https://github.com/netflix/chaosmonkey

[63] H. Wu, J. Pan, and P. Huang, "Efficient exposure of partial failure bugs in distributed systems with inferred abstract states," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1267–1283. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/wu-haoze

[64] P.-L. Wang, T.-W. Chao, C.-C. Wu, and H.-C. Hsiao, "Tool: An efficient and flexible simulator for byzantine fault-tolerant protocols," in *2022*

52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2022, pp. 287–294.

[65] C. Drăgoi, C. Enea, B. K. Ozkan, R. Majumdar, and F. Niksic, "Testing consensus implementations using communication closure," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428278

[66] S. Bano, A. Sonnino, A. Chursin, D. Perelman, Z. Li, A. Ching, and D. Malkhi, "Twins: Bft systems made robust," 2022. [Online]. Available: https://arxiv.org/abs/2004.10617

[67] L. N. Winter, F. Buse, D. de Graaf, K. von Gleissenthall, and B. Kulahcioglu Ozkan, "Randomized testing of byzantine fault tolerant algorithms," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586053

[68] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, "Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2517–2532.

[69] K. Li, P. Joshi, A. Gupta, and M. K. Ganai, "Reprolite: A lightweight tool to quickly reproduce hard system bugs," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–13. [Online]. Available: https://doi.org/10.1145/2670979.2671004

[70] "User statically-defined tracing." accessed: 2025-09-10. [Online]. Available: https://docs.ebpf.io/linux/concepts/usdt/

[71] "Aspectj." accessed: 2025-09-10. [Online]. Available: https://eclipse.dev/aspectj/

A Artifact Appendix

This artifact includes Rose implementation, accompanied by the experiments used to evaluate Rose, as seen in §6.

A.1 Abstract

The Rose artifact is divided in two components, i) Rose implementation as described in this paper, and ii) a suite of applications, including the required configuration files, patches, workloads, and scripts for its evaluation.

A.2 Description & Requirements

A.2.1 How to access. The artifact is available at https://github.com/sebastiaoamaro/rose and as a DOI at https://doi.org/10.5281/zenodo.18862553.

A.2.2 Hardware dependencies. A machine with more than 16 cores and 180 GB of available disk space.

A.2.3 Software dependencies.

- Linux (Tested for Ubuntu 24.04 LTS, kernel version 6.11)
- Vagrant (Tested for version 2.4.9)
- Python (Tested for version 3.12.3)
- VirtualBox (Tested for 7.1.16)

A.2.4 Benchmarks. The list of buggy traces necessary for bug reproduction is available at: bug_reproductions/traces/ inside the repository.

A.3 Set-up

To download Rose and all experiments, run:

```
git clone https://github.com/sebastiaoamaro/rose
cd rose && git submodule update --init --recursive
```

Before running the scripts, verify that you meet the requirements. This script will check and install accordingly.

```
cd artifact_evaluation && ./requirements.sh
```

To build and prepare the environment to test Rose run:

```
./prepare_vms.sh
```

The full process takes about 2 hours. We suggest using tmux, and leaving the build process running in the background.

A.4 Evaluation workflow [3]

A.4.1 Major Claims.

- *(C1): Reproduced Bugs.* Rose is able to reproduce the bugs described in Table 1. The results should match the ones in the table, except for time, schedules generated, number of total runs, and faults removed, since these can vary due to inherent machine variance. Nevertheless, they should be similar to the values in the table.
- *(C2): Tracer Overhead.* Rose tracer has a sub 3% overhead, and performs better than the other tracing options, when it comes to events generated, saved events, memory usage, processing time, and % overhead. The values in the table should be similar to the ones in Table 2
- *(C3): Heuristic Effectiveness.* By employing the function frequency heuristic, the number of functions detected reduces by 3 to 4 orders of magnitude, achieving a 99% reduction. Results should be similar to Table 3. The fault removal heuristic significantly reduces the search space by up to 96%. The values are displayed in Experiment 1 A.4.2 and should be similar to Table 1, although some variance is expected.

A.4.2 Experiments. *Experiment (E1): [Reproduced Bugs] [5 human minutes + 20 to 30 compute-hours]: This experiment showcases all of the bug reproductions done by Rose, the results are used to generate Table 1. We recommend that you run this experiment under tmux.*

[Execution] To run Rose to reproduce the bugs.

```
./reproduce_bugs.sh
```

[Results]

```
./display_bug_table.sh
```

[3] Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://sysartifacts.github.io/eurosys2026/.

A table containing all the bugs reproduced in this paper should be displayed in the terminal.

Experiment (E2): [Tracing Overhead] [5 human minutes + 2 to 3 compute-hours]: This experiment compares Rose tracing performance to other possible options. The evaluated metrics are events generated, events saved, trace size (memory usage), time to process the trace, and finally the % overhead. This experiment is used to generate Table 2. Since this is a perfomance comparison it is necessary to have no interference in the machine while running this experiment.

[Execution] To run the tests which evaluate the tracing component of Rose.

```
./reproduce_tracing_tests.sh
```

[Results]

```
./display_tracing_table.sh
```

Experiment (E3): [Heuristics Effectiveness] [5 human minutes + 1 compute-hour]: This experiment evaluates the impact of the function frequenty heuristic in discarding functions events. It is used to generate Table 3.

[Execution] To run tests which evaluate the effectiveness of Rose function frequency heuristic.

```
./reproduce_heuristics_tests.sh
```

[Results]

```
./display_heuristics_table.sh
```