Efficient Reproduction of Fault-Induced Failures in Distributed Systems with Feedback-Driven Fault Injection

Jia Pan*
Johns Hopkins University

Haoze Wu*
Johns Hopkins University

Tanakorn Leesatapornwongsa
Microsoft Research

Suman Nath
Microsoft Research

Peng Huang
University of Michigan

Abstract

Debugging a failure usually requires reproducing it first. This can be hard for failures in production distributed systems, where bugs are exposed only by some unusual faulty events. While fault injection testing becomes popular, existing solutions are designed for bug finding. They are ine□ective and ine□cient to reproduce a specific failure during debugging.

We explore a new type of fault injection technique for quickly reproducing a given fault-induced production failure in distributed systems. We present a tool, Anduril, that uses static causal analysis and a novel feedback-driven algorithm to quickly *search* the enormous fault space for the root-cause fault and timing. We evaluate Anduril on 22 *real-world* complex fault-induced failures from five large-scale distributed systems. Anduril reproduced all failures by identifying and injecting the root-cause faults at the right time, in a median of 8 minutes.

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

Keywords: failure reproduction; fault injection; distributed systems; debugging

ACM Reference Format:
Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, and Peng Huang. 2024. E□cient Reproduction of Fault-Induced Failures in Distributed Systems with Feedback-Driven Fault Injection. In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3694715.3695979

* Contributed equally to this work

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695979

1 Introduction

Failures in distributed system are notoriously di□cult to debug. Reproducing a failure is often the first step in debugging. An e□ective reproduction should recreate not only the superficial symptom but also the exact buggy workflow to help developers pinpoint the root cause. The failure reproduction should also be quick to enable timely fix and minimize downtime. Achieving both properties is di□cult due to the limited information available. A study of production distributed systems found that "developers spend a vast majority of the resolution time (69%) on reproducing the failure" [59].

Reproduction is particularly challenging for production failures triggered by external faults, such as disk error, network fault, and a broken dependent service. These *fault-induced failures* are common in large distributed systems, which have many dependencies on fault-prone hardware and software [18, 30]. Such a failure cannot be reproduced *even after developers succeed in constructing a suitable workload*, because the bug is only triggered by a specific fault at specific timing.

Specific faults can be triggered with fault injection tools; however, existing solutions are designed for testing and achieving good coverage, rather than for reproducing a specific failure. Some existing tools rely on users to specify faults to inject [22, 26, 36] or target certain types of fault-handling bugs [31]. They are not suitable for reproducing failures whose root-cause faults are not known yet. Other tools can iterate through possible faults systematically [10, 12, 17, 35, 47, 53] or randomly [6]. However, the space of possible faults can be prohibitively large in distributed systems as faults can happen at numerous execution points and each point can be executed many times. Therefore, it is not surprising that they can hardly inject the exact root-cause fault and/or timing.

Prior e□orts on failure reproduction focus on deriving the inputs or thread interleaving [5, 29, 54, 55]. Deterministic replay [20, 28, 32, 51] can faithfully replay a multi-threaded failure execution, but with expensive modification to production environment and with high performance overheads. Pensieve [59] is a non-intrusive tool to construct a workload, consisting of a sequence of external APIs, based on a failure log. These solutions cannot reproduce the fault-induced failures due to the lack of proper fault events. Simply combining such solutions with existing fault injection tools will face similar ine□ciency limitations as described earlier, because fault injection now becomes the bottleneck.

In this paper, we explore a novel fault injection technique that is tailored for reproducing a specific fault-induced failure in production distributed systems. We aim for both faithfully reproducing the buggy workflow and doing so quickly. The key challenge is to identify the root-cause fault in a large space of possible faults and their timing. This is hard even when the given failure log contains relevant exceptions. Production logs contain abundant noisy error messages. Many exceptions are also transient or do not cause immediate failure [24, 34]. Simply injecting exceptions in the failure log is insufficient or inefficient to reproduce complex fault-induced failures.

Our key insight is that many faults have correlated effects to the system behavior, so we can use the runtime information from one injected fault to estimate how likely it is for its related faults to reproduce the target failure, without actually injecting them. In utilizing this insight, our main contributions are to define what runtime information can capture this correlation, how to use it to rank unexplored faults by their likelihood of reproducing the failure, and how to update the ranking after each fault injection. This is challenging because we need to reason about the dynamic effect of a fault and determine the likelihood that this fault will reproduce the failure, *without actually executing the workload and injecting that fault*.

We devise a novel feedback algorithm to overcome this challenge. The algorithm starts with assuming that each of the selected log messages is important and computes priorities for the fault sites based on their causal connections to the messages. It then iteratively injects a high-priority fault, and if it does not reproduce the failure, uses the runtime information from that unsuccessful injection to re-rank the remaining faults before continuing with the next iteration.

To further improve the reproduction inefficiency, we use static analysis to prune fault sites that are irrelevant to a given failure. Nevertheless, since static analysis techniques are known to incur inaccuracies when applying to distributed systems, we only perform basic, conservative analysis. The static analysis step is designed as an optimization, while our dynamic exploration with feedback is the core.

We implement our solution in a tool called Anduril and evaluate it on 22 *real-world* fault-induced failures that we randomly sample from five large-scale distributed systems. Some evaluated failures took expert developers days to reproduce. Anduril successfully reproduced all the failures by identifying and injecting the root cause faults at the right timings, in a median time of 8 minutes. In comparison, the state-of-the-art solutions reproduced only 4 of the 22 failures respectively, and took $6\times$ - $280\times$ longer time. Interestingly, for five evaluated failures, Anduril's reproduction results identify new root causes that are overlooked in developers' manual analyses, and reveal flaws in the original patches. In three of them, the issues exist in the latest versions, and developers confirmed our findings to be new bugs.

The main contributions of this work are as follows:

- We explore a new fault injection technique for reproducing a specific fault-induced failure in distributed systems.
- We design and implement Anduril, a targeted fault injection tool that uses a combination of novel dynamic feedback algorithms and static analysis to quickly search the large fault space for the root-cause fault and timing.
- We evaluate Anduril on complex fault-induced failures, and show that it efficiently reproduce these failures.

Anduril is available at https://github.com/OrderLab/Anduril.

2 Background and Motivation

Problem Statement. We consider the problem of reproducing a production failure of a system $\mathcal{S}$ caused by unhandled or poorly-handled faults. We call such failures as *fault-induced failures*. Our problem takes the following as inputs:

(1) $\mathcal{S}$'s code that we can analyze, instrument, and run *offline*.
(2) a failure log file from production, where the deployed $\mathcal{S}$ is *not* instrumented by us. It may contain many irrelevant log messages and we do not expect the user to filter them.
(3) a driving workload, which can be any that triggers the fault location, not necessarily the exact production workload trace. There are several sources to obtain the workload: (a) reuse existing tests. Mature systems usually have tests with good coverage that are likely to exercise the affected feature; (b) leverage tools that sample online workload [4] or generate inputs [59] automatically; (c) construct a workload based on the symptoms. For example, if the symptom is the replication thread getting stuck, a developer can construct a write workload that exercises replication. This is often the first step for a developer trying to reproduce a failure. Anduril fits in this workflow and gets invoked when the developer has succeeded in workload construction but still cannot reproduce the symptom.
(4) a user-defined *failure oracle*, which encapsulates the key failure *symptoms*, such as a specific log message, a stacktrace that may or may not be in the log file, or an external state such as a corrupted data file. Our definition of *failure reproduction* is with respect to the oracle: the failure is reproduced if the oracle is satisfied.

Given the above inputs, our goal is to efficiently identify a *root-cause fault* that, when injected at a specific system execution point under the workload, can satisfy the oracle. Since the failure is fault-induced, injecting the root-cause fault is critical for the reproduction—the oracle is not satisfied by executing the workload without faults or with a wrong fault.

Scope. We target systems written in languages such as Java or .NET that capture faults as exceptions; hence faults are injected by throwing the relevant exception. We focus on failures with a single *root-cause* exception, which are common in practice. It is reflected in our evaluation datasets, which are user-reported failures that require extensive developer efforts to troubleshoot. To further confirm the significance, we conducted an empirical study. Specifically, we randomly sampled 50 failure cases from

Figure 1. Simplified code for a real HBase incident [21].

Zookeeper, HDFS, HBase, Kafka, Cassandra and analyzed them to check if the failures were induced by external faults, and if so, how many external faults were involved. We found that external faults triggered 41 of the 50 failures, and 39 of these 41 failures were induced by only one fault. Only two failures involved more than one root-cause fault.

2.1 A Motivating Example

We illustrate the challenges of reproducing fault-induced failures with a real-world incident in HBase [21]. A user reported that the HBase region servers (RS) in their cluster got stuck for several hours. The user examined the log file and found a `TimeOutException` warning when the RS tried to write a flush marker in the Write Ahead Log (WAL). The user also checked the stack trace and found that the WAL consumer thread was still alive. This is puzzling, because the consumer thread, if alive, should sync the WAL append request to HDFS and the `TimeoutException` should not occur. To debug this failure, developers had to reproduce it. But after constructing a workload, developers still could not reproduce the symptom.

This failure is likely fault-induced, so we try to reproduce it with `Anduril`. Since WAL rolling (switching to a new WAL file) is common in HBase, we use an *existing test* in HBase (`TestReplicationSmallTests`) as the workload. We use an oracle that indicates successful reproduction when the log file contains the timeout exception warning and the stack trace shows that the log roller is stuck at `waitForSafePoint`, as observed by the user. `Anduril` successfully reproduces the failure and identifies the root-cause fault. Its finding is consistent with what developers found out **after 19 days of digging and discussions with the user**.

HBase uses HDFS to store and access WAL files. To tolerate intermittent HDFS failures, developers designed a recoverable stream that would break upon faults and notify the upper layer to roll the writer and create a new stream. However, there exists a rare and fatal fault situation: (1) HBase creates the WAL file successfully; (2) The stream to HDFS breaks temporarily; (3) All subsequent asynchronous WAL entry appends by the consumer fail and move to `unackedAppends` (a queue) for retry. Moreover, the entries in `unackedAppends` exceed the batch size; unackedAppends queue keeps track of the append to WAL requests that are being processed by the underlying HDFS. (4) HBase creates a new writer and a new stream; (5) With the new stream, HBase retries the failed append entries and removes the acked ones from `unackedAppends` in `sync()`. Since only batchsize entries can be appended in one sync, it needs multiple rounds; (6) However, just at this time, a log roller or RS shutdown calls `waitForSafePoint`. Then, as Figure 1 shows, `unackedAppends` needs more than one `consume()` to empty, so the consumer gets stuck in a stale state with `waitForSafePoint`. It neither signals the condition nor does the sync in any later invocation and cannot recover. Even RS shutdown gets stuck at `waitForSafePoint`.

`Anduril` reproduces the failure by injecting the root-cause `IOException` at the right call site of `channelRead0` (line 34 in Figure 1) at the right timing described above.

This example shows several challenges in reproducing fault-induced failures. First, *fault injection is necessary to reproduce the failure*. Simply executing a workload that creates the WAL file and then writes to it does not trigger the failure. This explains why the existing tests did not expose the failure.

Second, *the space of possible faults and their timings can be very large*. Different versions of HBase we experimented with contain 18 K–28 K static *fault sites*, *i.e.*, code locations that can throw exceptions. Many of these fault sites execute multiple times. This makes the space of static and dynamic fault injection sites very large. For example, the developer-provided workload that triggers the above HBase failure executes 1 K+ static fault sites 208 K+ times. Moreover, the root cause fault site is executed 1 K+ times and *only 2 of these dynamic instances would satisfy the oracle*. Exhaustively trying each of them is prohibitively expensive since each injection needs to be accompanied by an execution of the workload.

Finally, *the root-cause fault and timing may not be readily identified by the logs.* The logs from the above failure might suggest killing the single consumer thread when it is appending a WAL entry, so that the other side receives no response and throws a `TimeOutException`. But this fault must be injected only when a WAL file is being rolled over—the log rolling would not get stuck by injecting the fault at another time. Even after figuring out the proper timing, we have only reproduced the superficial symptom but not the underlying root cause fault: the stale state that prevents the consumer from resuming the log rolling or retrying the failed WAL appends.

To faithfully reproduce what happened, we need to further go back to other code regions responsible for manipulating WALs. There are many such static and dynamic execution points across different functions. The fault must happen in the RPCs to HDFS when writing WAL appends, and should not

happen in the creation or closing of the file. Additionally, it should cause the failed appends to exceed the batchSize before the log is rolled. An experienced developer may eventually figure out the precise fault and its timing after manual reasoning about a vast amount of code and repeated experiment attempts. This process is extremely time-consuming and error-prone.

In practice, a complex failure can take experienced developers many weeks to reproduce [44]. Existing fault injection tools do not help much: given the large number of possible faults and their timings, it would take them a long time, if not impossible, to faithfully recreate the buggy execution.

2.2 Our Observation and Key Idea

We treat the task of failure reproduction as a search problem, where the goal is to *quickly* find the root-cause fault of a given failure from the space of all possible fault candidates. The space of all faults can be prohibitively large, and injecting a fault while executing the workload can be expensive; therefore, exhaustively exploring the space is not practical.

Although ideal, reproducing a production failure in a single attempt is usually hard due to the limited information available. Our key idea is to use the feedback from one failed injection to prune or deprioritize other *similar* faults without actually injecting them. Our insight is that the e□ects of multiple faults are often correlated—*i.e.*, they cause similar program behavior, such as executing the same catch block and printing the same log. In such a case, if injecting one fault does not reproduce the target failure, we can conclude that the other fault that produces similar program behavior is unlikely to reproduce the failure, *without actually injecting the fault*. This allows us to explore the fault space in multiple rounds: in each round, we inject a fault that is likely to reproduce the failure; if the attempt is not successful, we use the resulting program behavior as feedback to deprioritize some of the unexplored faults that are likely to generate a similar program behavior.

Note that our idea of feedback-driven fault injection is di□erent from the feedback used in fuzzing testing [60]. These techniques use feedback to improve code coverage for finding more bugs, while we use feedback to identify faults that are most likely to reproduce a target failure. The distinctions require fundamentally di□erent feedback designs.

3 Overview of ANDURIL

Anduril is a fault injection tool with the unique ability to reproduce a specific fault-induced failure that occurred in production. It aims to not only successfully reproduce the failure, but also achieve so quickly. Whether a given failure is fault-induced or not may be unknown. After developers make best e□orts in constructing workloads and observe that the failure still could not be reproduced, they can assume that the failure is likely fault-induced and invoke Anduril.

At a high level, Anduril achieves e□cient search in a large fault injection space through two levels of techniques: (1) static reasoning of the system code to deduce the *potentially* causal

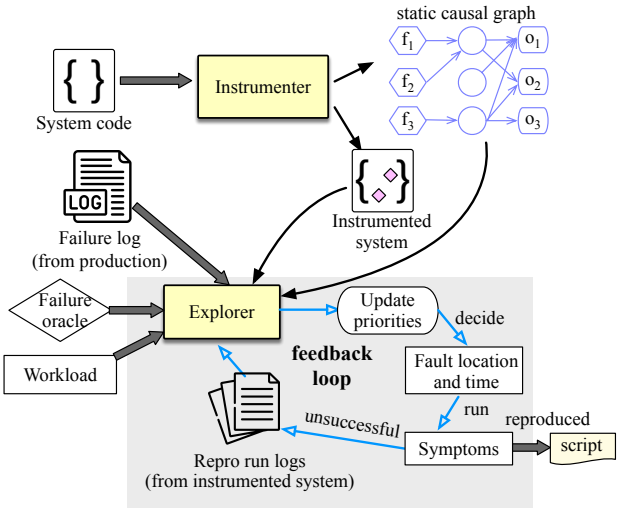


Figure 2. Anduril's architecture and workflow.

fault candidates for a given failure; (2) dynamic adjustment of the priorities for the fault candidates during the injection experiment. The former produces a conservative and imperfect *static causal graph*. The latter is key to Anduril.

Workflow. Anduril contains two major components, Instrumenter and Explorer (Figure 2). Instrumenter receives the system bytecode and the failure log file from production. It performs static analysis and computes a causal graph consisting of program points potentially related to the failure symptom. In addition, it inserts code snippets into the system for (1) injecting a fault to throw a desired exception, and (2) logging additional information to facilitate the feedback algorithms. Note that Anduril does not instrument the production system. Its input failure log is from the uninstrumented system.

Explorer's goal is to *quickly* search for the location of the root-cause fault as well as the timing for injection. It takes as inputs the static causal graph, a workload, the failure log, and an oracle. Explorer proceeds as follows:

1. Run the workload on the instrumented system without injecting any fault to generate a log file. Other than system's log messages, the file will also contain all the fault candidate instances that are exercised by the workload because Instrumenter injects those logging.
2. Compute initial priorities of all fault candidate instances based on the causal graph, the log file from step 1, and the failure log (from production system w/o instrumentation).
3. Pick the highest-priority candidate instance, run the workload, and inject the fault accordingly.
4. Invoke the oracle to check if the failure is reproduced.
   4.a If so, generate a script that *deterministically* injects the root-cause fault and reproduces the symptom.
   4.b If not, re-compute the priorities based on the logs from the failed attempt and causal graph. Go back to step 3.
5. If all possible faults are explored or a user-specified limit is reached, report that the failure cannot be reproduced.

Efficiency. Efficiency is measured by two metrics: (1) absolute time from step 1 to 4.a, which includes inherent costs of running the workload; (2) number of *rounds*, one round being from step 3 to 4, which reflects the speed of feedback.

Defining Feedback. An important question is *how to construct the feedback from an unsuccessful injection*. Our intuition is to extract the effect of a fault on the program's execution, which we call a *fault's traits*. When a fault is injected and it cannot reproduce the failure, we compare its traits with the other faults' traits. If they are similar, the other faults might not reproduce the failure either, so we can deprioritize them.

While a full execution trace (sequence of instructions, memory state, *etc.*) can serve as fault traits, acquiring it requires intrusive recording, which slows down the experiment and can distort the execution. Moreover, we would need to try a real injection for each fault, which contradicts our goal.

Instead, we use lightweight *observables* of an execution to determine the fault traits. We specifically choose log messages as observables for several reasons. First, during the exploration, we can easily collect them. Second, the execution of a distributed system node can be abstracted as a state machine, and it is a common practice for developers to log when a node enters a new state. Third, we can statically estimate the log messages that an *unexplored* fault can cause. The challenge for Anduril is to maximize the feedback from those observables, which contain limited information and can be noisy.

Assumptions. Anduril relies on the assumption that the system logs enough information to distinguish faulty and non-faulty executions. This is common in real-world production systems, where developers use discriminative logs, which are unique to failures, to help troubleshooting. Anduril is not the only solution that depends on the quality of logging. Many other debugging and reproduction solutions, such as Pensieve [59], also require some useful log messages. Anduril does *not* assume that the log messages reveal the exact fault causing a failure; such failures are easy to reproduce.

Anduril currently performs a single injection in each round, so it only target failures with a single root-cause fault. A system may fail due to an unhandled fault, along with other faults that are tolerated by the built-in error handling mechanism. Since these handled faults do not contribute to the failure, Anduril can still reproduce such a failure. Furthermore, if a root-cause fault causes multiple exceptions, Anduril will also reproduce these exceptions and the failure, as long as it injects the root-cause fault correctly. The failures that Anduril cannot reproduce are those that need multiple root-cause faults that do not have causal dependencies.

To use Anduril, a developer does not need to know if the failure is caused by a single root-cause fault. She can first apply Anduril since single-fault-induced failures are common. If Anduril fails to reproduce the symptoms, the failure may be caused by multiple faults. In such cases, Anduril may produce logs close to production failure logs. These logs can guide developers to apply Anduril iteratively, fixing one fault at a time in the workload and rerunning Anduril.

A target failure may be caused by a concurrency bug compounded with a fault. For such cases, Anduril assumes that the workload provided enforces the specific thread interleaving in which the root cause fault sites are traversed and focuses on deducing the correct fault under the schedule to reproduce the failure. How to deduce the specific thread interleaving for a concurrency bug is an orthogonal problem and still actively researched [23, 41, 45]. A future work direction is to develop solutions that explore the space of thread interleaving and the space of faults together to reproduce such failures.

4 Instrumenter

In principle, Anduril can explore all possible faults in the target system, and rely on its dynamic feedback loop to identify the root-cause fault. However, this is wasteful, because not all faults are related to the failure we try to reproduce. Therefore, Anduril starts with a subset of faults.

4.1 Computing Causal Graph

To determine the subset of faults to explore, Anduril computes a *static causal graph* for given observables, *i.e.*, a list of log messages. We will explain in § 5.1 how we derive this list. For each observable, Anduril identifies what *fault sites*—program points that can throw exceptions—are causally related to it. A fault site is causally related to an observable if an exception at that site can result in the observable to appear. Just extracting exceptions in the failure log is insufficient because the system may not log an exception that it think can be handled but that turns out to be the root cause.

A standard static approach performs complete data-flow analyses, e.g., computing the program dependence graph [15] or program slices [52] from a logging statement. For large distributed systems, such analyses are expensive. Moreover, they can miss complex causal relations common in large systems, *e.g.*, due to dependencies from callbacks, third-party libraries, external modules, RPCs, etc. To address this issue, Anduril uses the jumping strategy proposed in Pensieve [59]. For example, given `if (x==y)`, we directly search for program points across functions that write to `x` or `y`, and treat them to be possibly causal. Although this strategy can produce false dependencies, it captures complex dependencies that would otherwise be missed. Such a trade-off matches Anduril's design of relying on its dynamic feedback algorithm to identify the root-cause fault from a large space of possible faults.

Causal Analysis. Using this strategy, we compute causality for a node (program statement) s by recursively identifying the *causally prior* nodes for s depending the node type. We extend the algorithm in Pensieve in two important ways. First, we add to it exception flow analysis that is crucial to reason about root causes faults. Second, instead of generating one chain, we derive many chains and combine them into a DAG where source nodes represent fault sites and sink nodes represent

statements that produce a given list of log messages. The following three node types are similar to the ones in Pensieve.

• A *location* node represents a program point being executed. We compute its causally prior nodes by using control-flow analysis to find its dominators, which may be a *condition*, an *invocation*, or a *handler* node. We represent program points that generate the initial list of log messages as *location* nodes.

• A *condition* node represents a program point executed that requires the satisfaction of a boolean expression. Its causally prior nodes are computed in two ways. First, we consider the condition node as a location node, and compute its causally prior node as described above. Second, we use a slicing analysis to find the location nodes that can potentially satisfy the boolean expression. The extracted nodes may include *location* nodes, *condition* nodes, and *invocation* nodes.

• An *invocation* node represents the program execution reaching a method invocation statement. Its causally prior nodes are computed by a call-graph analysis that finds location nodes representing the code locations that invoke this method.

Exception Analysis. Anduril introduces four new nodes.

• A *handler* node represents reaching the entry point of an exception handler (*i.e.*, `catch` block). Its causally prior nodes are computed by finding the locations that throw certain exceptions caught by this handler. We develop an interprocedural exception analysis that computes 1) for each method, what exceptions could be thrown from the method and from what locations; 2) for each local variable carrying the value of an exception, the potential types of this exception, the data flow of this variable, and the location where the variable is thrown.

In addition, our exception analysis handles cross-thread exception propagation due to asynchronous programming common in distributed systems. For example, in Java, a thread may wrap a task in a `Callable`, submit it to a new thread, and later wait for the returned `Future` to finish. If something wrong happens in the task execution, the waiting thread would get an `ExecutionException`, but the underlying fault is inside the scheduled task. Anduril analyzes the inner scheduled code, according to the future semantics, to augment the control and data flow of the exceptions for the causal graph.

The computed causally prior nodes may be *location* nodes and three kinds of *exception* nodes described below.

• An *internal-exception* node represents an exception being thrown by an invocation to an internal method that is implemented by the system, but that method is not the origin of the thrown exception. The exception being thrown is propagated from invocations within the internal method. For example, an invocation to an internal method `process` throws a `SocketException`, because `process` invokes another internal method `send`, which in turns makes a library call that throws this exception. We distinguish an *internal-exception* node, because although it has an exception thrown, it is only propagating the exception. Treating it as a fault site to inject would be superficial. We need to continue the causal analysis.

Algorithm 1 Build static causal graph

Input: System code S; a list of log messages L
Output: Causal graph $G = (V, E)$

```
1  prog_stmts ← Map(S, L)
2  q ← NewNodes(prog_stmts, LOCATION)
3  while q ≠ ∅ do
4    node ← q.pop()
5    if node.type ∈ {NEW-EXCPT, EXT-EXCPT} then
6      continue
7    cps ← CausallyPrior(S, node)
8    for c ∈ cps do
9      E.add({c, node})
10     if c ∉ V then
11       V.add(c)
12       q.add(c)
```

```
1  private void flush0(...) {
2    try {
3      if (buffer.size() > 0) {
4        // instrumented by FIR to trace the fault site.
5        FIR.traceSite(fid, ...);
6        // instrumented by FIR for fault injection.
7        FIR.throwIfEnabled(fid, occurrence, ...);
8        out.write(buffer, 0, buffer.size());
9      }...
10   } catch (IOException e) {
11     future.completeExceptionally(e);
12   }
13 }
```

Figure 3. Anduril instruments two kinds of code.

• A *new-exception* node represents an exception being thrown from an internal system method and that method is the creator of the thrown exception, *i.e.*, the internal method uses a `throw new` statement to create an exception that is uncaught. However, if this new exception is thrown because of an external exception (defined below), we downgrade it to an internal exception, because we aim to find the deeper root cause.

• A *external-exception* node represents an exception thrown by a method from standard or third-party libraries. Both *new-exception* and *external-exception* nodes are the fault sites we are looking for. Our recursive analysis of finding causally prior nodes stops when encountering the two types of nodes.

Causal Graph. Algorithm 1 shows how Anduril constructs the static causal graph. The function `CausallyPrior()` at line 7 computes all the causally prior nodes for a given node as explained before. Given the complexities of distributed systems we target and that our static analysis is conservative, the computed causal graph can be large. For the real motivating example in HBase (§ 2.1), the computed causal graph contains 357,816 vertices and 868,373 edges.

4.2 Adding Injection and Logging Code

In addition to computing the static causal graph, Anduril's Instrumenter additionally instruments the target system for Explorer to intercept the fault sites. The instrumentation is

applied on the program points corresponding to each source node in the computed causal graph, *i.e.*, a fault site.

Anduril instruments two kinds of code snippets: injection and tracing code. When the instrumented injection code is reached, it calls the Explorer runtime, which determines if an exception should be thrown. The tracing code collects runtime information including the time and occurrences of the fault site, which will be used by the feedback algorithm (§5.2.3). Figure 3 shows an example. Right before the fault site `out.write` is executed, the instrumented `traceSite()` (line 5) call will record information about this site, after which the instrumented site `throwExceptionIfEnabled` (line 7) will check if Explorer wants to try any fault candidates here and throws an exception correspondingly.

5 Explorer

In this section, we describe Anduril's core designs for feedback-driven fault injection.

5.1 Identifying Relevant Observables

For e□cient exploration, we first identify relevant observables. Many error or warning messages in the failure log may not be related to the target failure. Worse, some error or warning messages may appear even when the system runs successfully—such messages indicate errors that are handled by the system. We do not expect the user to specify which log messages are relevant (the oracle only tells the failure symptoms). Anduril automatically derives relevant observables.

5.1.1 Method Anduril runs the workload to obtain a *normal log file*, and then compares it with the failure log file. It assumes that *any* log message that only appears in the failure log is a relevant observable. This assumption significantly increases the size of the fault space, since Anduril will then consider all fault sites causally related to those observables; but this is essential to maximize the chance that the fault space includes the root-cause fault. Anduril will identify and de-prioritize irrelevant fault sites through its feedback loop.

Simply using a standard di□ for the log comparison does not work. Log messages usually contain timestamps, making them appear unique. It is also common for distributed system log messages to interleave across runs. Thus, Anduril groups the messages in each log file by thread name. It then sanitizes the log entries to remove the timestamps. Next, Anduril applies the Myers di□erence algorithm [42] between the sanitized logs with the same thread name. If the failure log includes threads not present in the normal log, we include all log messages from those threads as relevant observables. This per-thread di□ works well because developers usually explicitly name threads uniquely to ease log-driven debugging.

5.1.2 Usage Relevant observables are used in two steps of the Anduril workflow. First, they are used by Instrumenter in computing the static causal graph. Recall that the graph construction requires a list of log messages as input (§ 4.1). Using relevant observables, instead of all messages, is essentially an optimization to reduce the causal graph. To get the normal log file needed in the log comparison algorithm, Anduril runs the given workload once before the fault injection starts.

Algorithm 2 Update priorities of observables

Input: Failure log f_log; Initial normal log n_log; a list of log files from each injection round run_logs

```
relevant_observables ← Compare(n_log, f_log)
for log ∈ run_logs do
    missing ← Compare(log, f_log)
    present ← relevant_observables − missing
    for observable ∈ present do
        observable.priority = observable.priority + 1
```

Second, relevant observables are updated in each fault injection round. When a fault injection is unsuccessful, that round naturally produces a normal log file. Anduril then redo the log comparison algorithm on the new log file to get the updated list of relevant observables. It will be used in the feedback algorithm which will be covered in the next section.

An important property is that the relevant observables computed in the first step will be a *superset* for the relevant observables in the second step, because the provided failure log file is fixed. While the normal log file may change in each round and produce unseen messages compared to the failure log, only messages in the failure log are our target. Anduril leverages this property to avoid computing a causal graph in each round, and is able to use the causal graph computed in the first step throughout the experiment.

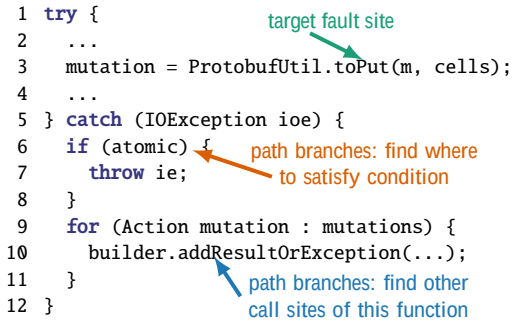
5.2 Feedback Algorithm

Explorer considers prospective faults represented by the source nodes in the static causal graph. It computes feedback for each prospective fault to prioritize the exploration. This section describes our feedback algorithm and its considerations.

5.2.1 Updating Feedback As § 3 explains, Anduril constructs feedback based on fault traits—a fault's connections to the relevant observables. We thus start by considering the observables of an unsuccessful injection (recall that relevant observables are updated after each trial). When an injection fails to reproduce the target failure, we look at the observables that we expected to see but did not. These missing observables may be related to the failure, so we want to find faults that could cause them. This is how Anduril updates its feedback.

To do this, Anduril assigns a priority I_k for each relevant observable o_k. We use smaller values to present higher priorities. Initially, all I_k are zero. When an injection is unsuccessful, Anduril compares (§ 5.1) the run log with the failure log to determine which observables in the causal graph appear in the current log and which ones are missing. It then prioritizes the missing messages in the following rounds, so it increments the I_k for each present message by one (Algorithm 2).

5.2.2 Updating Fault Site Priority With the updated feedback, we next determine the priority F_i of each fault site f_i (a




```
1  try {
2    ...
3    mutation = ProtobufUtil.toPut(m, cells);
4    ...
5  } catch (IOException ioe) {
6    if (atomic) {
7      throw ie;
8    }
9    for (Action mutation : mutations) {
10     builder.addResultOrException(...);
11   }
12 }
```

Figure 4. Inaccuracies accumulate along static causal paths.

source node in the causal graph), based on the observables that f_i can possibly cause, *i.e.*, a path from f_i to o_k exists in the causal graph. F_i reflects how likely f_i will cause the failure.

One approach is to simply set the fault priority to the (maximum) priority of the observable(s) it can cause. However, this makes Explorer suffer from inefficiencies due to the nature of static analyses. When going from a fault to a log message in the causal graph, inaccuracies inevitably accumulate. For example, in Figure 4, HBase-19876 has the symptom log within the method call at line 10, and this method is also called in other 30+ locations. We will waste multiple attempts to exclude irrelevant fault sites. In addition, when Instrumenter reaches line 10 in building the causal graph, it needs to create two path branches: for the `catch` at line 5, and the condition check at line 6. Consequently, there will be injection attempts that prioritize to satisfy the condition at line 6, while the root-cause fault site is at line 3 in the `try` block. However, we cannot easily prune these irrelevant branches that cause the inefficient exploration. They are worth considering because the workload *confirms* most of them. Some injections do expose the call to the method at line 10 via other callers. Some injections also do have the condition at line 6 satisfied.

Therefore, it is not certain that a fault will produce all the observables it has paths to in the causal graph. The further an observable on a path is, the more uncertain it will be produced.

Motivated by this uncertainty property, Anduril considers the **the spatial distance** $L_{i,k}$ from the fault f_i to its observables o_k in the causal graph, and defines F_i to be a function of I_k and $L_{i,k}$, which will be described in § 5.2.4.

5.2.3 Updating Fault Instance Priority So far, f_i refers to only *static* fault candidates, which are the exception type and its location in the code. A fault site can be exercised more than once, and we observe that reproducing complex failures often require a specific exception at specific state—injecting it too soon or too late do not reproduce the failure. To refer to an instance of a fault candidate here, we will use the notation $f_{i,j}$ to refer to the j-*th occurrence* of the fault site f_i.

Thus, reproducing a failure requires considering all runtime instances of the fault candidates, but this is infeasible in practice. The number of instances can be large. We need to intelligently choose which instances to explore.

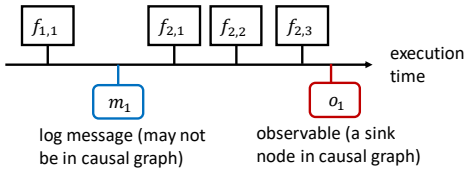


Figure 5. An execution timeline with different fault instances.

Anduril determines the fault instance priority based on the **temporal distance** $T_{i,j,k}$ between a fault instance $f_{i,j}$ to an observable o_k that we want to trigger. Our intuition is similar to prioritizing fault sites: the further a fault is from the message, the more uncertain it will trigger that message.

To obtain $T_{i,j,k}$, a straightforward choice is the absolute time distance. However, absolute time varies significantly across runs, even when running the system on the same machines. Moreover, it is very sensitive (e.g., in a millisecond scale, a little difference can greatly affect the priority).

We instead determine $T_{i,j,k}$ based on *logical* time distance. In particular, we count the number of log messages between $f_{i,j}$ and o_k. To explain its rationale, consider an example timeline in Figure 5. One option of defining logical time is by the *order* of the fault instance. For example, the fault instances $f_{1,1}$, $f_{2,1}$, $f_{2,2}$, and $f_{2,3}$ could cause observable o_1. Using order, we can assign $T_{1,1,1} = 3$, $T_{2,1,1} = 2$, $T_{2,2,1} = 1$, and $T_{2,3,1} = 0$. This choice addresses the problems of the absolute time. However, it focuses too much on the fault f_2. If f_2 turns out to be irrelevant, we have to try all three instances of it before trying f_1. Using number of log messages instead, $T_{1,1,1} = 1$ (from m_1), $T_{2,1,1} = 0$, $T_{2,2,1} = 0$, and $T_{2,3,1} = 1$, which avoids putting too much penalty on f_1. Our intuition is that the presence of a log message indicates a possible state change in the system; distance by the number of messages indicates state changes to reach the observable, which can be more robust than the relative order of fault instances.

To calculate $T_{i,j,k}$, one obstacle is that we do not know how the fault instances are distributed in the failure log timeline, because this log is from production run. We also can not directly calculate it from the experiment logs, which likely miss o_k (we want to trigger it). To address this issue, we collect the fault instance distribution when obtaining the first normal log. We then use an alignment algorithm based on the longest common subsequence to map the fault instances from the normal log's timeline to the production failure log's timeline. Specifically, from § 5.1, we get a set of matched log entries. By pairing neighbors of size 2, we get the finest matched intervals between normal and failure logs. Then we scale the distribution of fault instances in one interval in the normal log into its counterpart interval in the failure log.

5.2.4 Putting All Priorities Together We now have the feedback on observables, the spatial priority for each fault site, and the time (occurrence) priority for each fault instance. How to put these terms together? We observe how an experienced developer would approach the problem of reproducing complex failures. They often divide and conquer: first decide the

most promising fault site, and then decide the most promising time of that fault site. We follow the same approach.

Anduril first focuses on selecting the high-priority fault site. At this stage, we do not consider the time information. Distributed systems have high concurrencies, so even a few milliseconds can exercise many fault sites. Time priorities would not well differentiate them. For fault site f_i, we consider all observables that f_i reaches in the causal graph. Let $p_{i,k}$ denote the partial priority of f_i when considering its causal observable o_k. Then the full priority F_i will be some aggregate of $p_{i,k}$. We pick $\min_k(p_{i,k})$[1], so that we maximize the chance to reproduce one observable in one injection run. An alternative is $\sum_k(p_{i,k})$, which means we try to trigger all o_k with the best possible fault. Each $p_{i,k}$ may have different magnitudes, so the summation can be less sensitive to the effect of feedback compared with using min. Since $p_{i,k}$ is positively related to both I_k (priority of o_k) and $L_{i,k}$ (distance from f_i to o_k), either adding or multiplying them would be reasonable. Because some observables are noisy, we do want small perturbations to alleviate their effect. Thus, the final $F_i = \min_k(L_{i,k} + I_k)$.

Next, we focus on selecting the high-priority instance of a fault site. At this stage, it is intuitive that we utilize $T_{i,j,k}$ to rank over all N instances of the same fault injection site. Thus, $F_{i,j}$, the priority of $f_{i,j}$, is represented as $\min_{0 \le j \le N, o_k} T_{i,j,k}$. Note that o_k is determined in the first stage when selecting F_i. Also, it is fine for o_k to change during the feedback. At that time, we would utilize different k for calculating $F_{i,j}$.

Then at each trial, Anduril chooses the fault instance $f_{i,j}$, such that both $\min_i(F_i)$ and $\min_j(F_{i,j})$ are satisfied.

5.2.5 Flexible Priority Window In theory, with the priorities computed, Anduril should inject the fault with the highest priority. However, distributed systems are non-deterministic, so the highest-priority fault might not occur in a round, causing that round to be wasted (no fault is injected).

To address this issue, Anduril uses a *flexible-window* selection scheme. Instead of only picking the highest priority, Anduril considers the top k highest priority fault candidates. In each round, if *any* of the candidates in the window occurs, Anduril will inject that fault candidate, even if it does not have the highest priority, and remove that candidate from the window. If no fault is injected in a round, Anduril doubles the window size k for the next round to consider more candidates. With n fault candidates, there are at most $O(\log n)$ rounds without any injections, effectively reducing wasted rounds.

6 Limitations

Anduril is not effective in the following scenarios: (1) the target failure is caused by delays or silent error codes that do not throw exceptions; (2) the target failure is caused by multiple, causally independent root-cause faults; (3) the given workload does not exercise the code path that contains the fault site; (4) the logs produced by the system are insufficient to differentiate between a faulty and non-faulty execution;[2]

Distributed systems have internal concurrency, which leads to the challenge that a fault injected too soon or too late may not reproduce the failure. The feedback algorithm in Anduril, specifically the fault instance priority update (§ 5.2.3), is precisely designed to address this challenge. The internal concurrency may reorder the log messages across different reproduction runs. Anduril can handle this with its per-thread log diff method (§ 5.1.1). However, if the concurrency causes crucial log messages to disappear, the reproduction success becomes probabilistic. To improve the chances, we can run Anduril multiple times per round and use the combined logs.

When a target failure is caused by concurrency bugs compounded with a fault, Anduril assumes that thread interleaving has been deduced and enforced in the workload. If this assumption does not hold, Anduril would not be able to deterministically reproduce the failure.

Anduril only finds true bugs that cause the given symptom, *i.e.*, satisfy the failure oracle. In theory, a given symptom can result from multiple bugs. In these rare cases, Anduril's "reproduction" may report a bug that differs from the one that occurred in production. This is also a useful bug discovery.[3]

Our static causal analysis is neither sound nor complete. For example, it would miss the data flow across disk and reflection. However, the analysis is designed to be conservative so it prioritizes soundness while compromising on completeness, relying on dynamic feedback to handle false dependencies.

7 Implementation

We implemented Anduril mainly in Java. Anduril currently works with distributed systems in JVM bytecode, which supports many distributed systems written in Java, Scala, and Clojure. However, the key ideas of Anduril are applicable to other high-level languages that capture a fault as a program exception. We built Anduril's Instrumenter on the Soot static analysis framework [50] with around 3,700 SLOC. Anduril Explorer is written with around 5,600 SLOC.

Anduril uses log messages as the observables. We need to properly parse the messages for mapping them to code in the causal graph construction, and for comparing and analyzing them to compute feedback. We implement a log parser in Scala for this purpose. The parser supports common logging conventions (e.g., using Log4j). If a system uses non-standard logging formats, users need to provide the format configuration (regular expressions) for our parser to identify the key fields. Such a configuration is a one-time and easy effort. For the five systems we evaluated, we only used two configurations: one for Kafka and a second for the other four systems.

[1] min priority value represents highest priority

[2] The five systems we evaluate in § 8 generate sufficiently discriminating logs; Anduril could use the logs as clues to reproduce *all* the failures we tried.

[3] We found 5 new bugs like this, as shown in § 8.2.

System	LOC	Fault Site		
		Total	Inferred	Dynamic
ZooKeeper	120 K–176 K	6,225	572	2,878
HDFS	351 K–1,187 K	20,803	4,761	73,186
HBase	211 K–1,649 K	24,497	2,905	106,095
Kafka	166 K–201 K	45,109	1,134	423,298
Cassandra	152 K–307 K	5,899	1,258	2,022,819

Table 1. Lines of code of the target systems and the fault sites for the 22 failures, which occurred in different system versions. The fault site results represent the mean. *Total*: all static fault sites in the system. *Inferred*: static fault sites Anduril identifies by its causal graph algorithm. *Dynamic*: occurrences of the *inferred* fault sites.

To efficiently implement Anduril's feedback loop, we make several optimizations. In each round, Anduril calculates the priority for each injection before running the workload. This calculation is expensive. In some cases, due to the large number of injection candidates, it takes 5 minutes to calculate the priorities, while the workload itself only takes about 40 seconds. We observe that part of information can be calculated beforehand. Specifically, the fault site priority is based on the summation of the causal graph distance and feedback on observables. We thus pre-calculate the distances before the experiment and query them in each round. Also, based on the design of our core algorithm, we only store the information of instances with highest time priority for each fault site, which significantly reduces the time priority table serialization costs. After each round, we need to compute the log diff for updating feedback, which is expensive. We implement the diff algorithm in C to reduce the overhead.

8 Evaluation

Our evaluation addresses the following questions: (1) can Anduril reproduce known fault-induced failures (§ 8.1 and § 8.1)? (2) what is the impact of our feedback algorithm (§ 8.3)? (3) how does Anduril compare with existing solutions (§ 8.4)? (4) how sensitive is the feedback algorithm (§ 8.5)? (5) how fast are the decision and static analysis (§ 8.6)?

Failure dataset. To show Anduril's practical efficacy, we reproduce real-world failures in five large distributed systems: ZooKeeper, HDFS, HBase, Kafka, and Cassandra. We search their JIRA trackers for fault-related issues and collect 40 cases that need non-trivial reasoning to reproduce. We randomly choose 22 of them and report all our attempts with Anduril, without excluding any of its failures. These failures are difficult to reproduce manually, even for experienced developers.

Table 1 lists the code size and the number of fault sites for the target systems. Our appendix lists the description for each failure as well as the fault required to reproduce the failure.

Since these failures are resolved, we know the ground-truth to construct the failure oracle. For the failure log, if the original ticket does not provide a log file, we manually reproduce the failure first based on the ground truth to obtain the log. In real usage, Anduril directly uses the failure log from production.

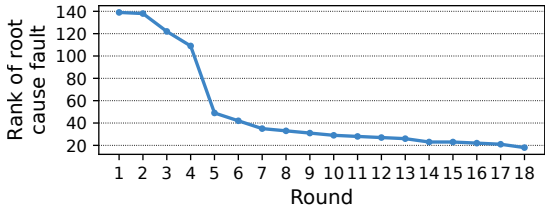


Figure 6. The rank of the root-cause fault site for HBase-25905. The rank improves across trials due to the feedback Anduril computes.

We did not apply Anduril on non-fault-induced failures in the evaluation. If Anduril is applied on a non fault-induced failure, it would run until the specified limit (e.g., 2000 trials) is reached and conclude that it could not reproduce the failure.

Workload generation. We used the target systems' existing tests (created before the failures) as the workloads for 13 failures. For 6 other failures, we used the workloads that developers constructed and provided with the issue tickets. We created workloads for the remaining 3 failures from the descriptions or logs. Two of them were easy to construct, while the third took us more time. This is because we took time to realize that the root-cause fault instance only appears under a specific thread interleaving, and subsequently updated the workload to enforce the thread interleaving. Our workloads averaged 185 lines of code. Note that in all cases, the workload alone could not trigger the failure without fault injection. We used the same workloads for Anduril and the baselines to compare the fault injection.

Other experimental setup. We set the initial window size k (§ 5.2.5) to 10 for all failures. The experiments are conducted on servers running Ubuntu 18.04 with 20-core 2.20 GHz CPU and 64 GB memory.

8.1 Efficacy of Failure Reproduction

Table 2 shows the failure reproduction results. Anduril successfully identified the root-cause fault and reproduced the failure for *all* 22 issues, which demonstrates its capabilities.

Besides effectiveness, Anduril aims to reproduce failures efficiently. We measure efficiency by both absolute time and the injection rounds. In reproducing 12 cases, Anduril takes less than 10 minutes and fewer than 20 rounds. The median efficiency for the 22 failures is 8 minutes and 11 rounds. The longest time Anduril took is 445 minutes (281 rounds), which is still acceptable compared to days or even weeks that it often takes for developers to manually reproduce complex failures. Six of our collected failure tickets contain information about when developers reproduced the failure, the median time for manual failure reproduction is 136 hours.

Anduril is effective on both failures with relatively smaller fault space (e.g., ZooKeeper-4203) and complex failures with enormous fault space (e.g., HBase-25905, HDFS-12070).

HBase-25905. This is the motivating example. Anduril automatically identifies 53 relevant observables (§5.1). Some show that normal HRegion flush failed because of `TimeOutException` waiting for consumer's sync. Some are due to the `IOException`

Failure		FIR												SOTA Solutions			
		Full Feedback		Exhaustive Fault Instance		Fault-Site Distance		Fault-Site Dis. w/ instance limit		Fault-Site Feedback		Multiply Feedback		FATE		CrashTuner	
		Rnd.	Time	Rnd.	Time	Rnd.	Time	Rnd.	Time	Rnd.	Time	Rnd.	Time	Rnd.	Time	Rnd.	Time
ZK-2247	(f1)	5	2 min	535	95 min	203	21 min	74	11 min	8	2 min	23	6 min	-	-	271	41 min
ZK-3157	(f2)	3	2 min	2548	872 min	428	107 min	136	39 min	29	10 min	1	1 min	55	13 min	34	23 min
ZK-4203	(f3)	10	3 min	454	74 min	135	28 min	65	12 min	65	4 min	8	2 min	-	-	4114	840 min
ZK-3006	(f4)	13	2 min	2696	430 min	151	7 min	13	2 min	4	1 min	1	1 min	-	-	135	17 min
HD-4233	(f5)	30	25 min	-	-	-	-	68	60 min	89	92 min	34	16 min	-	-	-	-
HD-12248	(f6)	16	8 min	-	-	-	-	175	60 min	374	73 min	-	-	2781	304 min	-	-
HD-12070	(f7)	9	5 min	-	-	3207	592 min	-	-	-	-	10	5 min	1925	357 min	-	-
HD-13039	(f8)	6	17 min	-	-	-	-	160	346 min	19	37 min	17	31 min	-	-	-	-
HD-16332	(f9)	8	5 min	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HD-14333	(f10)	11	7 min	-	-	-	-	-	-	64	20 min	1	1 min	-	-	-	-
HD-15032	(f11)	74	236 min	-	-	-	-	-	-	-	-	-	-	-	-	-	-
HB-18137	(f12)	4	11 min	-	-	-	-	-	-	-	-	10	45 min	-	-	-	-
HB-19608	(f13)	19	44 min	-	-	-	-	-	-	645	189 min	176	124 min	-	-	-	-
HB-19876	(f14)	258	191 min	-	-	-	-	-	-	-	-	226	206 min	-	-	-	-
HB-20583	(f15)	15	44 min	-	-	-	-	-	-	11	7 min	77	223 min	-	-	-	-
HB-16144	(f16)	281	445 min	-	-	-	-	-	-	-	-	476	643 min	-	-	-	-
HB-25905	(f17)	18	93 min	-	-	-	-	-	-	-	-	-	-	-	-	-	-
KA-12508	(f18)	8	8 min	1565	640 min	-	-	153	87 min	108	50 min	-	-	-	-	-	-
KA-9374	(f19)	7	7 min	1585	514 min	-	-	283	301 min	57	32 min	-	-	-	-	-	-
KA-10048	(f20)	3	8 min	-	-	-	-	-	-	-	-	-	-	-	-	-	-
C*-17663	(f21)	2	4 min	-	-	-	-	-	-	-	-	1	2 min	-	-	-	-
C*-6415	(f22)	17	99 min	-	-	-	-	-	-	-	-	1	10 min	-	-	-	-

Table 2. E□cacy on reproducing 22 *real-world* failures with Anduril, its variants, and two state-of-the-art solutions. Rnd.: number of fault injection rounds to reproduce the failure. "-" means a failure cannot be reproduced after running for 24 hours. The columns under Anduril other than *Full Feedback* use alternative designs within Anduril (§ 8.3 explains their meanings). We also evaluated a stacktrace injector (§ 8.4).

showing the underlying HDFS stream is broken during consumer's sync. There are also noisy logs showing transient failures in receiving blocks in `DFSClient`.

In dynamic experiment, at first the rank of the root-cause site is over 130. During this process, the fault that trigger failure in transferring blocks is injected and also, the `IllegalStateException` is injected to directly kill the consumer and we got the `TimeOutException` in flushing. Gradually, Anduril would think that the log that is related to broken HDFS stream should take higher priority, as shown in Figure 6. (location+feedback works here) Finally, at trial 18, we inject in `getRPCResults` with the corresponding occurrence that triggers the failed stream of WAL writers and we got the expected symptom. What is more, only 2 out of over 1000 instances of the root-cause site can satisfy the oracle.

ZooKeeper-4203. When the servers start the leader election, an `IOException` occurs while the leader is accepting the socket from a follower. Due to defective design, this fault fails the whole leader election service and no more followers can join the quorum. Anduril identified three relevant observables, including a symptom message about the socket service failure. The causal graph analysis finds around 1,000 fault site candidates. The root-cause fault site gets assigned a high priority that ranks 8th initially. Then the feedback algorithm improves the ranking from 8 to 6 within four rounds. Finally, Anduril reproduces the failure in 10 rounds.

HBase-16144. One regionserver that holds the replication queue's lock aborted due to an unknown transient failure. Afterward, no other regionserver could claim the queue to do synchronization. Anduril automatically infers 78 potentially relevant logs. The causal graph Anduril contains 3075 fault sites, one of which is the root-cause fault. From the evaluation result, it turns out to be the most challenging case for Anduril.

The reason is that the single message close to the root cause only shows that a regionserver aborts but not why. It is a common practice in HBase to abort the regionserver when encountering a failure. From our static analysis, Anduril infers that more than 2500 fault sites are causally related to the ABORT message. At the first round, the root-cause fault site's rank is 342. Through feedback, Anduril quickly learns the importance of the ABORT log and prioritizes faults that are causally related. Although most of them could indeed trigger the abort, only a tiny subset can satisfy the oracle. Those unsuccessful injections cause Anduril to doubt the importance of the ABORT log. Overall, the feedback still works and promotes the rank of the site to 187 at best. At this time, our two-level combination design (§5.2.4) enacts: for each fault site, it just tries the one that is close on timeline to the ABORT message and traverses the fault sites e□ciently without wasting too much time on the same fault site. Finally, Anduril reproduces the failure in round 281.

Param.		Failure Id																					
		f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15	f16	f17	f18	f19	f20	f21	f22
initial k	=1	10	1	14	1	28	24	1	6	14	5	99	1	12	715	6	468	22	13	10	1	1	24
	=3	8	2	13	2	34	19	1	3	11	3	148	2	13	257	8	316	20	9	9	1	1	17
	=10	5	3	10	13	30	16	9	6	8	11	74	4	19	258	15	281	18	8	7	3	2	17
adj. s	+1	5	3	10	13	30	16	9	6	8	11	74	4	19	258	15	281	18	8	7	3	2	17
	+2	5	3	11	13	43	72	8	5	13	11	63	3	19	257	14	283	19	9	7	3	2	79
	+10	5	3	5	13	44	649	8	4	13	13	67	2	18	257	-	384	-	8	7	2	2	116

Table 3. Sensitivity of two key parameters in Anduril. The initial window size k for the flexible priority selection scheme (§ 5.2.5). adj. s: The priority value adjustment for observables (§ 5.2.1). The highlighted rows are results for the default settings.

8.2 Enhancing Expert's Diagnosis and Patch

A given failure symptom can result from multiple bugs. Interestingly, for 5 of the reproduced cases, Anduril's results allow us to discover other root causes. For example, in one case, the original root cause developer diagnosed was message loss caused by network I/O faults, which then caused the snapshot repair to be blocked forever, but Anduril identifies a deeper root cause where an early-stage disk I/O fault causes the target keyspace to not be created at all. Note that Anduril's new root cause also leads to failure symptoms that satisfy the oracle, but it is deeper in the root cause chain.

A tangible impact of Anduril's findings is that they expose flaws in the patches developers wrote. In four cases, the original patches do not work. For the above example, the original patch only uses some retry logic to deal with message loss, which could not fix the situation that the target keyspace does not exist. In one case, although the original patch works, we develop a new patch that is more e□cient. We submitted our findings. In 3 cases, the issues exist in the latest versions, and **developers confirmed them to be new bugs**. We also received confirmation for the more e□cient patch we developed in a fourth case. We omit the bug ids here for double-blind review.

8.3 Importance of Techniques

To measure the importance of di□erent techniques in Anduril, we conduct an ablation study that removes or replaces certain components in Anduril. In particular, we implement and evaluate five di□erent strategies shown in Table 2.

The *exhaustive* strategy only leverages the Anduril static causal graph and tries all the instances from the fault sites in the causal graph. The *fault-site distance* strategy sets a fault site's priority only as the graph distance term $L_{i,k}$ without any feedback. It also applies the flexible priority window (§ 5.2.5). The *fault-site distance with instance limit* similarly only uses $L_{i,k}$ for priority and additionally considers only the first 3 instances of each fault site. The *fault-site feedback* strategy additionally includes the feedback of observables, I_k, but it it does not distinguish priorities for fault candidate instances (i.e., no $T_{i,k}$). It also applies the 3-instance limit. The *multiply feedback* strategy uses both the fault site priority F_i and fault instance priority $F_{i,k}$, but it simply uses $F_i \times F_{i,k}$ to combine them instead of using our two-level approach (§ 5.2.4).

As Table 2 shows, the complete Anduril significantly outperforms all five variant strategies. The best variant, *multiply feedback*, only reproduces 15 failures in 88 minutes and 71 rounds on average. By comparing the variants, we can also see the importance of each technique. For example, while the *exhaustive* variant leverages the Anduril causal graph to prune many fault sites (Table 1), allowing it to outperform non-Anduril solutions, its result is much worse compared to the other variants. This suggests that dynamic feedback is crucial. Comparing among the dynamic variants suggests that each feedback consideration plays an important role. The *multiply feedback* result suggests that the two-level approach is superior to simple combination of di□erent priorities.

8.4 Comparison with Other Solutions

CrashTuner [35] and FATE [19]. We compare Anduril with CrashTuner and FATE, two state-of-the-art fault injection solutions that target distributed systems. CrashTuner proposes to use *meta-info variables* to identify critical timings for injecting faults. FATE employs the notion of failure IDs to avoid redundant fault injections and uses prioritization to explore new failure scenarios first.

As Table 2 shows, CrashTuner and FATE only reproduce four and three failures, respectively. They are designed for bug finding instead of failure reproduction, thus they focus on improving coverage, which cause them to waste significant time exploring faults that are irrelevant to a specific failure.

StackTrace-injector. Additionally, we implement a *stacktrace-injector* for comparison. It extracts all warning and error messages in the failure log, and parses the fault sites in those messages as well as the stack traces if logged. During experiment, it only injects if the executed site is one of the logged sites and the stack trace matches the failure log.

The stacktrace-injector only reproduces 9 failures in 78 minutes and 230 rounds on average. It can perform well if the failure log is clean and the root-cause fault appears in the log. For example, it can reproduce Kafka-12508 in the first round, because only two fault sites are extracted from the log, one of them being the root-cause site. However, when the root-cause fault does not appear in the failure log, it cannot reproduce the failure. Also, if the log contains irrelevant error messages or when the root-cause fault site is executed frequently, it

System	Inject. Req.		Round Init.	Workload
	Cnt.	Latency		
ZooKeeper	2,955	2 μs	6.7 s	6.4 s
HDFS	36,091	0.30 μs	22 s	12 s
HBase	31,162	0.40 μs	25 s	27 s
Kafa	423,298	0.20 μs	41 s	10 s
Cassandra	2,022,819	29.10 μs	124 s	25 s

Table 4. Median injection requests received by Anduril Explorer, the latency for each decision, the median initialization time for each injection round, and the workload time.

performs poorly. For example, it extracts 9 static fault sites from the log for HDFS-15032, but takes 1839 rounds and 634 minutes to reproduce the failure. Note that the input failure log we use is created from a test workload and thus small. But in real deployment, the failure log directly comes from production and will be much larger and more noisy.

Pensieve [59]. Pensieve can e□ciently reproduce failures for distributed systems. However, it is not publicly available for us to conduct comparisons. More importantly, Pensieve focuses on deducing the inputs for a failure, which is an orthogonal problem. Combining Pensieve with other fault injection tools would su□er from the same limitations and yield similar results shown above. From Table 2, we can approximate the performance of extending Pensive to support exception events. The exhaustive strategy relies on static causal reasoning of exceptions, which performs poorly in our evaluated failures compared to using full dynamic feedback in Anduril.

8.5 Sensitivity of Key Parameters

Anduril's feedback algorithm has two key parameters: the initial priority window size k (§ 5.2.5), and the priority value adjustment s for observables' feedback. We use their default settings ($k = 10$, s increment by one) in the experiments. We evaluate Anduril's sensitivity to di□erent settings. Table 3 shows the results. The feedback algorithm is overall robust to di□erent settings, under which Anduril still reproduces most of the 22 failures. However, they do result in relatively small di□erences for most cases.

8.6 Performance

We evaluate the performance of the Explorer. In each injection round, Anduril needs to compute and update di□erent priority factors, thus there is an initialization cost for each round. Initially this cost was high, which caused some experiments to proceed slowly. We made several optimizations (§ 7) that greatly reduce the cost. Table 4 shows the median optimized initialization time for di□erent systems. We also measure the median number of injection requests Anduril receives for the evaluated failures as well as the median latency for Anduril to make a decision. As Table 4 shows, after Explorer finishes initialization, subsequent decisions in each round are fast.

We also measure the static analysis performance. The time ranges from 11 s to 344 s, depending on the code size and complexity. The most time-consuming step is analyzing the exceptions and handlers takes. But the longest time is only 162 s (HBase). The slicing analysis is fast, finishing within a few seconds. Our appendix shows the detailed results.

9 Related Work

Fault Injection Since distributed systems frequently encounter faults, fault injection testing is popular and has been extensively studied. Due to the large fault space, existing solutions often perform random injection [6] or rely on users to write the policies [1]. Recent solutions [3, 12, 17, 19, 27, 31, 35, 37, 39, 48, 49, 53] propose more advanced techniques to improve fault injection testing, such as using failure IDs [19], meta-info variables [35], and abstract states [53].

These solutions are designed for finding bugs. They focus on coverage and thus can be ine□cient in triggering a specific failure. To the best of our knowledge, Anduril is the first fault-injection tool designed for reproducing a given failure. Anduril directly searches for faults relevant to a given failure, and takes a feedback-driven approach to e□ciently pinpoint the root-cause fault and their timing.

Failure Reproduction Reproducing production failures is notoriously di□cult, motivating many solutions. An extensively explored technique is record and replay. By logging all sources of non-determinism at runtime, including input, thread scheduling, and environment interaction such as file and network I/O, it can replay a past execution. This technique is particularly useful for reproducing concurrency bugs. However, it is known to incur prohibitive runtime overhead. Despite significant e□orts [2, 13, 20, 28, 32, 33, 38, 40, 45, 46, 51], the overhead is still too high to apply on production systems. Anduril is non-intrusive. It does not instrument the production system or perform runtime recording. It only uses the existing failure log from the production run.

Symbolic execution is also used to reproduce failures by searching for an execution trace containing the desired symptoms. Given a coredump, ESD [55] extracts a subprogram using static slicing, and then uses symbolic execution to search for paths that exercise the entirety of this subprogram and reach the symptom. However, symbolic execution does not scale to large systems because of the path explosion problem.

Pensieve [59] proposes an event-chaining approach using static analyses to reproduce the input (sequence of external APIs) for a given failure. Its static analysis uses a jumping strategy that aggressively skips code paths. Anduril is inspired by Pensieve. However, it is a complementary e□ort. Anduril is a fault injection tool to reproduce fault-induced failures, while Pensieve focuses on input-induced failures. For example, Pensieve discards exception causal conditions because of its focus on input. In addition, Pensieve relies on static analyses, while the core of Anduril is dynamic feedback.

Fuzzing Fuzzing testing [7–9, 9, 16, 43, 57, 58] widely uses feedback to guide input generation. Its main goal is to generate

inputs to increase code coverage and uncover as many bugs as possible. Anduril is designed to reproduce a specific failure. Its feedback is therefore targeted to quickly identify the root-cause fault that can reproduce the specific failure. Thus, it requires a fundamentally di□erent feedback design.

There are some work [11, 25, 56] employing stacktrace to reproduce the bug. However, large-scale distributed systems are designed to be fault-tolerance and it is common that they will not log the stacktrace of the faults they suppose they can handle (although it can be the root cases of the failure sometimes). So Anduril does not employ stacktrace but instead construct causal graph to deduce the possible fault sites. What's more, we also construct the stacktrace baseline that purely utilize the stacktrace of the faults printed out in the logs. However, as the information in appendix shows, it does not perform well for most of the cases.

10 Conclusion

We presented Anduril, a fault injection tool designed to e□-ciently reproduce fault-induced failures in deployed distributed systems. Anduril uses a novel dynamic feedback-driven injection algorithm enhanced by a static causal reasoning step to pinpoint the root-cause fault and timing in a large fault space. We evaluated Anduril on real-world complex fault-induced failures in large distributed systems. Anduril quickly reproduced all the failures and outperformed existing solutions.

Acknowledgments

We thank the anonymous reviewers and our shepherd Aurojit Panda for their valuable and detailed feedback that improved our work. We thank CloudLab [14] for providing us with the experiment platform. This work was supported by NSF grants CNS-2317698, CNS-2317751, and CCF-2318937.

References

[1] Jepsen: a framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen.

[2] Gautam Altekar and Ion Stoica. ODR: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 193–206, New York, NY, USA, 2009. Association for Computing Machinery.

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

[4] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI '04, San Francisco, CA, December 2004. USENIX Association.

[5] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 362–371. IEEE, 2013.

[6] Cory Bennett and Ariel Tseitlin. Chaos monkey released into the wild. http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html, 2009.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[9] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. A systematic review of fuzzing techniques. *Computers & Security*, 75:118–137, 2018.

[10] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: consistency-guided fault injection for cloud systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 536–547, 2020.

[11] Ning Chen and Sunghun Kim. Star: Stack trace based automatic crash reproduction via symbolic execution. pages 198–220, 2015.

[12] Yinfang Chen, Xudong Sun, Suman Nath, Ze Yang, and Tianyin Xu. {Push-Button} reliability testing for {Cloud-Backed} applications with rainmaker. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1701–1716, 2023.

[13] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, OSDI '02, Boston, MA, December 2002.

[14] 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. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference*, USENIX ATC '19, pages 1–14, Renton, WA, jul 2019. USENIX Association.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[17] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST '17, page 149–165, USA, 2017. USENIX Association.

[18] Supriyo Ghosh, Manish Shetty, Chetan Bansal, and Suman Nath. How to fight production incidents? an empirical study on a large-scale cloud service. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 126–141, 2022.

[19] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 238–252, Berkeley, CA, USA, 2011. USENIX Association.

[20] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 193–208, USA, 2008. USENIX Association.

[21] Apache HBase. Shutdown of WAL stuck at waitforsafepoint. https://issues.apache.org/jira/browse/HBASE-25905, 2021.

[22] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[23] Je Huang, Charles Zhang, and Julian Dolby. Clap: recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 141–152, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS XVI, British Columbia, Canada, May 2017. ACM.

[25] Wei Jin and Alessandro Orso. Bugredux: reproducing field failures for in-house debugging. In *ICSE '12: Proceedings of the 34th International Conference on Software Engineering*, pages 474–484, 2012.

[26] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. PREFAIL: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 171–188, New York, NY, USA, 2011. ACM.

[27] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation*, NSDI '07. USENIX Association, April 2007.

[28] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: E cient online multiprocessor replay via speculation and external determinism. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 77–90, New York, NY, USA, 2010. Association for Computing Machinery.

[29] Tanakorn Leesatapornwongsa, Xiang Ren, and Suman Nath. Flakerepro: automated and e cient reproduction of concurrency-related flaky tests. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1509–1520, 2022.

[30] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. What bugs cause production cloud incidents? In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 155–162, 2019.

[31] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen 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*, ASPLOS '18, pages 419–431. ACM, 2018.

[32] Hongyu Liu, Sam Silvestro, Wei Wang, Chen Tian, and Tongping 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*, PLDI '18, page 344–358, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: E cient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 327–336, New York, NY, USA, 2011. Association for Computing Machinery.

[34] Chang Lou, Peng Huang, and Scott Smith. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 559–574, Santa Clara, CA, February 2020. USENIX Association.

[35] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 114–130, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Paul D Marinescu and George Candea. Lfi: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 379–388. IEEE, 2009.

[37] Paul D. Marinescu and George Candea. LFI: A practical and general library-level fault injector. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, DSN '09, pages 379–388, June 2009.

[38] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 693–708, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 388–402, New York, NY, USA, 2021. Association for Computing Machinery.

[40] Pablo Montesinos, Luis Ceze, and Josep Torrellas. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution e ciently. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, page 289–300, USA, 2008. IEEE Computer Society.

[41] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, page 267–280, USA, 2008. USENIX Association.

[42] Eugene W Myers. An o (nd) di erence algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[43] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802. IEEE, 2019.

[44] PagerDuty. The discovery of apache zookeeper's poison packet. https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet, 2015.

[45] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. PRES: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 177–192, New York, NY, USA, 2009. Association for Computing Machinery.

[46] Gilles Pokam, Cristiano Pereira, Shiliang Hu, Ali-Reza Adl-Tabatabai, Justin Gottschlich, Jungwoo Ha, and Youfeng Wu. CoreRacer: A practical memory race recorder for multicore x86 tso processors. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, page 216–225, New York, NY, USA, 2011. Association for Computing Machinery.

[47] Patrick Reynolds, Charles Killian, Janet L. Wiener, Je rey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI '06, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[48] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 143–159, Carlsbad, CA, July 2022. USENIX Association.

[49] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation*

(OSDI 22), pages 143–159, 2022.

[50] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[51] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. Doubleplay: Parallelizing sequential logging and replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, page 15–26, New York, NY, USA, 2011. Association for Computing Machinery.

[52] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449, 1981.

[53] Haoze Wu, Jia Pan, and Peng Huang. E□cient exposure of partial failure bugs in distributed systems with inferred abstract states. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation*, NSDI '24, pages 1267–1283, April 2024.

[54] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: Error diagnosis by connecting clues from run-time logs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[55] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 321–334, New York, NY, USA, 2010. Association for Computing Machinery.

[56] Cristian Zamfir and George Candea. Execution synthesis: a technique for automated software debugging. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 321–334, 2010.

[57] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[58] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. Ptfuzz: Guided fuzzing with processor trace feedback. *IEEE Access*, 6:37302–37313, 2018.

[59] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 19–33, New York, NY, USA, 2017. Association for Computing Machinery.

[60] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

A Evaluation Details

We include additional details and results that are omitted in the main paper due to space constraints. The content in this appendix is not peer-reviewed.

Reproduced failures. Table 5 lists the failures Anduril reproduced. It also shows the types of exceptions that Anduril injected to reproduce the failures.

Stacktrace-injector. We also implemented a baseline solution that injects faults based on stacktraces. Its results are listed in the last two columns in Table 5.

New root causes discovered. Anduril found new root causes for five failures that di□ered from the developers' diagnoses, as shown in Table 6. We also list the log messages that Anduril used to infer the new root causes. In four cases, developers' patches could not fix the scenarios that Anduril exposed.

Failure Id	old root cause	new root cause	exploited log	old patch works
ZK-4737	network issue in loading dataset	disk issue in loading dataset	root cause	no
HD-17157	disk failure causes meta data loss	network issue causes no response in second stage of block recovery	intermittent	yes
HB-28014	disk failure causes empty WAL	underlying HDFS issue causes failure of adding replication peers	root cause	no
KA-15339	delay in making connector	disk issue in appending records at startup	root cause	no
CA-18748	request/response loss of repair	disk issue in making column family	root cause	no

Table 6. The new root cause and flaw in patch Anduril discovered when reproducing the failures.

Static analysis. We applied Anduril's static causal graph algorithm to reduce the fault sites. Table 7 shows the time taken by the static analysis for each case. It also shows the break-downs of the static analysis time.

Runtime details. Table 8 shows the runtime details of the Anduril Explorer, such as the number of injection requests it received, the decision latency, the initialization time per round, and the workload time for each case.

Failure Id		Description	Injected Fault	Stacktrace injector	
				Rnd.	Time
ZK-2247	(f1)	Server unavailable when leader fails to write transaction log	IOException	6	1 min
ZK-3157	(f2)	Connection loss causes the client to fail	IOException	112	23 min
ZK-4203	(f3)	The leader election is stuck forever due to connection error	IOException	68	13 min
ZK-3006	(f4)	Invalid disk file content causes null pointer exception	IOException	31	2 min
HD-4233	(f5)	Rolling backup fails but the server keep serving	FileNotFoundException	-	-
HD-12248	(f6)	Exception when transfering file system image to namenode causes the namenode checkpointing to ignore the image backup	InterruptedException	-	-
HD-12070	(f7)	Files will remain open indefinitely if block recovery fails which creates a high risk of data loss	IOException	5	1 min
HD-13039	(f8)	Data block creation leaks socket on exception	IOException	-	-
HD-16332	(f9)	Missing handling of expired block token causes slow read	IOException	-	-
HD-14333	(f10)	Disk error during namenode registration causes datanodes fail to start	IOException	-	-
HD-15032	(f11)	Balancer crashes when it fails to contact an unavailable namenode	SocketException	11	22 min
HB-18137	(f12)	Empty WAL file causes Replication to get stuck	IOException	1	1 min
HB-19608	(f13)	Interrupted procedure mistakenly causes a failed state flag	IOException	-	-
HB-19876	(f14)	The exception happening in converting pb mutation messes up the CellScanner	IOException	-	-
HB-20583	(f15)	The failure during splitting log causes resubmit of another failed splitting task	IOException	-	-
HB-16144	(f16)	Replication queue's lock will live forever if regionserver acquiring the lock has died prematurely	IOException	-	-
HB-25905	(f17)	Transient namenode failure in HDFS causes WAL services in HBase to stop making any progress	IOException	-	-
KA-12508	(f18)	Emit-on-change tables lose updates after error and restart	IOException	1	1 min
KA-9374	(f19)	Blocked connectors disable the Workers	IOException	1839	634 min
KA-10048	(f20)	Consumer's failover under MM2 replication configuration causes data gap between 2 clusters	IOException	-	-
C*-17663	(f21)	Interrupted FileStreamTask compromise shared channel proxy	IOException	-	-
C*-6415	(f22)	Snapshot repair blocks forever if get no response of makeSnapshot	IOException	-	-

Table 5. The failures reproduced by Anduril, their brief descriptions, the types of the faults Anduril injects to reproduce the failures, and the time and the number of rounds Anduril takes to reproduce the failures.

Failure Id	LOC	Time			
		Exception	Slicing	Chaining	Total
ZooKeeper-2247	120K	0.9s	0.1s	0.06s	11s
ZooKeeper-3157	136K	0.9s	0.1s	0.07s	13s
ZooKeeper-4203	178K	2.0s	0.1s	0.1s	18s
ZooKeeper-3006	176K	1.5s	0.1s	0.1s	15s
HDFS-4233	351K	3.6s	0.1s	0.3s	31s
HDFS-12248	1M	103s	0.4s	6s	237s
HDFS-12070	940K	120s	0.7s	6s	228s
HDFS-13039	880K	46s	0.5s	1s	113s
HDFS-16332	1187K	165s	0.9s	3s	280s
HDFS-14333	1054K	140s	1.2s	9s	261s
HDFS-14333	1130K	143s	3.5s	3s	294s
HBase-18137	303K	186s	0.8s	25s	344s
HBase-19608	211K	156s	1.5s	1.4s	286s
HBase-19876	211K	169s	1.3s	1.5s	302s
HBase-20583	1649K	162s	1.3s	1.5s	303s
HBase-16144	1204K	95s	0.7s	1.4s	130s
Kafka-12508	201K	74s	1.1s	1s	179s
Kafka-9374	166K	37s	0.9s	0.7s	112s
Cassandra-17663	307K	42s	0.9s	1.5s	94s
Cassandra-6415	152K	4s	0.2s	0.3s	31s

Table 7. LOC: lines of code analyzed; **Exception**: time used in exception analysis; **Slicing**: time used in slicing analysis; **Chaining**: average time used in creating a causal chain for one observable.

Failure	Inject. Req.		Round	Workload
	Cnt.	Latency	Init.	
ZooKeeper-2247	1726	2 μs	6.6s	2.6s
ZooKeeper-3157	4444	2 μs	7.1s	10.2s
ZooKeeper-4203	1158	2 μs	6.1s	10.4s
ZooKeeper-3006	4184	2 μs	6.7s	0.9s
HDFS-4233	42753	11 μs	8.5s	64.1s
HDFS-12248	36091	0.2 μs	22s	12.0s
HDFS-12070	18588	0.3 μs	16s	9.4s
HDFS-13039	34587	4 μs	18s	95.1s
HDFS-16332	19182	0.4 μs	26s	10.4s
HDFS-14333	84583	0.2 μs	24s	3.0s
HDFS-15032	276518	0.2 μs	142s	12s
HBase-18137	344098	0.3 μs	93s	46.5s
HBase-19608	21028	0.4 μs	16s	17.3s
HBase-19876	31162	0.4 μs	18s	13.5s
HBase-20583	117406	0.1 μs	58s	27s
HBase-16144	16783	1 μs	25s	30s
Kafka-12508	49443	0.3 μs	41s	6.3s
Kafka-9374	797152	0.1 μs	40s	14.2s
Cassandra-17663	168825	0.2 μs	64s	33s
Cassandra-6415	3876813	58 μs	184s	17s

Table 8. Injection requests received by Anduril Explorer, the latency for each decision, the median initialization time for each injection round, and the workload time.