ExChain: Exception Dependency Analysis for Root Cause Diagnosis

Ao Li[1], Shan Lu[23], Suman Nath[2], Rohan Padhye[1], Vyas Sekar[1]
[1]Carnegie Mellon University, [2]Microsoft Research, [3]University of Chicago

Abstract

Many failures in large-scale online services stem from incorrect handling of exceptions. We focus on exception-handling failures characterized by three features that make them difficult to diagnose using classical techniques: (1) implicit dependencies across multiple exceptions due to state changes; (2) silent code handling without logging; and (3) separation (in code and in time) between the root cause exception and the failure manifestation. In this paper, we present the design and implementation of ExChain, a framework that helps developers diagnose such exception-dependent failures in test/canary deployment environments. ExChain constructs causal links between exceptions even in the presence of the aforementioned factors. Our key observation is that mishandled exceptions invariably modify critical system states, which impact downstream functions. A key challenge in tracking these states is balancing the tradeoff between performance overhead and accuracy. To this end, ExChain uses state-impact analysis to establish potential causal links between exceptions and uses a novel hybrid taint tracking approach for tracking state propagation. Using ExChain, we were able to successfully identify the root cause for 8 out of 11 reported subtle exception-dependent failures in 10 popular applications. ExChain significantly outperforms state-of-art approaches, while producing several orders of magnitude fewer false positives. ExChain also offers significantly better accuracy-performance tradeoffs relative to baseline static/dynamic analysis alternatives.

1 Introduction

Failures in large-scale production systems continue to be a significant source of frustration for developers and loss of customer satisfaction and revenues for service providers. A common root cause of system failures is *incorrect handling of exceptions* or errors.

Developers today can check whether a failure occurred during the handling of an exception and whether that exception was thrown during another exception handler, etc; i.e., track exception chains. Unfortunately, existing workflows are not useful for diagnosing failures whose root causes are outside the current exception chain, a type of failures that we refer to as **exception-dependent failure (EDFs)**. Exception-dependent failures involve multiple exceptions whose handling periods do *not* overlap (i.e., they do not belong to the same exception chain) and yet the (mis)handling of one *root cause* exception triggers a downstream exception which eventually leads to a failure.

In the context of large systems, diagnosing failures often necessitates an in-depth understanding of EDF. As highlighted by Yuan et al., "Almost all catastrophic failures (92%) stem from the incorrect management of non-fatal errors that are explicitly signaled in software" [59]. Complementing this, our manual analysis of 150 failures across multiple Apache Foundation projects affirms that 85% of these failures originate from exceptions. Unfortunately, existing solutions for root cause diagnosis of EDFs are insufficient; in our evaluation with 11 EDFs, state of the art slicing, log analysis, and statistical debugging techniques could identify the root causes of only three of fewer EDFs.

EDFs differ from simple exception-chains [25] in three key aspects that make them especially challenging to diagnose:

- *Implicit stateful dependencies:* In an explicit exception chain, the final failure-inducing exception is part of a cascaded chain of exceptions triggered by the root cause. In EDFs, however, the root cause exception can lead to a failure not only by just modifying the control flow of a program, but also by subtly changing the state.
- *Silent handling:* Due to the common practice of silent exception handling [59], the root cause exception may not be logged. This makes it difficult to diagnose exception-handling failures that may occur later on.
- *Spatial/temporal separation:* Finally, the root cause and the failure may be spatially and temporally distant from each other. For instance, the root cause exception may be triggered by one user request and the failure may be triggered by a different request that has some data dependency with the root cause.

For these reasons, we argue that it is critical to complement *exception chain* information, which is commonly produced and visualized by standard libraries in languages like Java and Python, with *exception dependency* information for effective failure diagnosis.

In this paper, we present ExChain, a tool that enables developers to diagnose EDFs by automatically inferring exception dependency. ExChain works by instrumenting the production system binary. At run time (e.g., during integration testing or canary deployment), ExChain routines, which were instrumented into the production binary, automatically log all exceptions and their contexts, and perform a dynamic-static hybrid analysis to identify *causal dependencies* among all exceptions; i.e., an exception e_2 causally depends on an exception e_1 if e_1 is responsible for e_2, either *explicitly* (e.g., throw e_2 inside a catch block of e_1) or *implicitly* (e.g., e_1 changes application states that causes e_2). Whenever a failure occurs, developers can query the exception-dependency graph produced by ExChain to see whether the failure was caused by the mishandling of an exception.

ExChain employs a set of novel state-impact analyses to establish potential causal links between exceptions. First, ExChain analyzes how exception-handling code changes program state—we call this *affected state analysis*—by identifying memory locations whose values are impacted by the change in control flow when compared to non-exceptional execution. Second, ExChain analyzes immediate causes for exceptions by tracking control flow backwards from the program locations where exceptions are raised and identifying memory locations whose values are responsible in triggering the exception—we call this *responsible state analysis*. Third, ExChain incorporates taint analysis to monitor the state affected by code addressing one exception as it influences values which activate other exceptions. Notably, ExChain introduces a hybrid algorithm, blending dynamic taint tracking for heap objects with static taint tracking for local primitives. This strategic approach positions ExChain uniquely in the taint-tracking arena, offering accuracy akin to dynamic taint analysis while achieving the overhead benefits of static taint tracking. Collectively, these methods enable ExChain to determine causality between exceptions, proving invaluable for EDF diagnosis.

We evaluated ExChain using 10 diverse applications from the Apache Foundation, spanning various domains and averaging 6K stars. Out of 11 reproducible EDFs instances, ExChain identified root causes for 8, outperforming the state-of-the-art statistical debugging, slicing, and log analysis tools, which pinpointed only 3 or fewer issues.[1] In performance metrics, ExChain introduced an average latency overhead of 8%, half attributable to its techniques and half to underlying JVM tools – tools we aim to optimize in future versions. When juxtaposed with an alternative design employing static taint tracking, ExChain was a mere 2% costlier but detected 5 more root causes. In contrast, while dynamic taint tracking identified the root cause for all failures, it was found to introduce a substantial performance overhead, reaching up to 50 times. This suggests that while accuracy is crucial, the

```
class PageProvider {
    int counter;
    Page[] cache;
    void processRequest(Request req) {
        Page page = cache[req.pageIndex];
        try {
            if (page.isAttached()) {
                logAndThrow(
                    "attached page.");
            }
            page.resolve(req.sessionId);
            counter++;
            // render page
        } catch (StalePageException e1) {
            // StalePageException is
            // swallowed and not logged.
            page.refresh(counter);
        } catch (InitError e2) {
            throw new FatalError(e2);
        }
    }
    void logAndThrow(String msg) {
        logger.info(msg);
        throw new InitError(msg);
    }
}
class Page {
    int sessionId;
    void resolve(int sessionId) {
        attach();
        if (sessionId != this.sessionId) {
            throw new StalePageException();
        }
        ...
        detach();
    }
}
```

Req. 1

Req. 2

HTTP 200

HTTP 500

The method does not detach the page when a StalePageException is thrown.

Figure 1: Wicket [7] fails to detach page when exception occurs [56]. Green circles represents the first request that triggers a `StalePageException` and leaves the page attached. Red squares represents the second request which triggers the `InitError` and causes the system to fail.

accompanying performance trade-off can be significant. The results show that ExChain's hybrid taint-tracking presents a useful accuracy-overhead trade-off: an accuracy closer to dynamic taint tracking with an overhead closer to static taint tracking. This makes ExChain suitable for test or canary deployments where the modest overhead is acceptable for the ease of failure diagnosis.

The source code of ExChain is available at: https://github.com/aoli-al/exchain.

2 Motivation

In this section, we present an example to illustrate the notion of exception-dependent failures and discuss why they are an important class of problems that do not yet have good solutions in practice.

2.1 A Motivating Example

We present a simplified real-world example to illustrate

[1] ExChain could have identified the root causes of two additional failures with a better static taint tracking tool than what our current prototype uses.

exception-dependent failures (EDFs) and the unique challenges in diagnosing their root causes. To keep the discussion short, we pick a simple application and keep details to a minimum, noting that similar problems also manifest in more complex and popular applications such as Hadoop, HDFS, and Tomcat (as we show in §6).

Figure 1 shows a simplified code snippet from the Apache Wicket web server [7], a popular choice for dynamic web applications. When a user requests a page, Wicket returns the cached version of the page for better performance (Line 5). Next, Wicket resolves the page based on the `req.sessionId` (Line 11). This `sessionId` allows Wicket to differentiate between requests from various users. Specifically, if the `sessionId` associated with a rendered page differs from a new request, Wicket initiates re-rendering for the page. This process is implemented in the `resolve` method. It begins by attaching the page object to the current context (Line 30), ensuring it remains exclusive to the current requester and safeguarded against accidental modifications by concurrent requests. Upon completion, the method detaches the page object. The `resolve` compares the `sessionId` of the page object with that of the requester. In cases of mismatch, it throws a `StalePageException` (Line 32) so that the `PageProvider` will refresh the page (Line 17).

Unfortunately, the `resolve` method in Wicket 9.4.0 has a bug that can result in an HTTP 500 error response. Figure 2 illustrates the sequence of requests that trigger this bug. The issue involves two requests. The first request asks for a stale page, i.e., the `req.sessionId` does not match the current `page.sessionId`, which causes a `StalePageException` at Line 31. Due to this exception, the `resolve` method fails to detach the `page` object, leaving it attached to the current context. The exception is later swallowed and the page is silently refreshed in Line 17. The second request asks for the same page requested by the previous request, and the `cache` array returns correctly. However, since page is still attached, `processRequest` throws an `InitError` exception that eventually causes the HTTP 500 error response. The reporter of the above Wicket issue spent significant time to identify the root cause and to understand how it is causally related to the failure [56].

Note that the final failure (HTTP response 500) causally depends on one or more exceptions. The dependency can be explicit or implicit. An exception e_j *explicitly* depends on another exception e_i if e_i's catch block explicitly throws e_j. In the Wicket example above, `FatalError` explicitly depends on `InitError`. On the other hand, e_j *implicitly* depends on e_i if e_i changes application state in a way that causes e_j; i.e., there is a data-flow between the effect of e_i and the cause of e_j. In the Wicket example, `InitError` implicitly depends on `StalePageException` since the latter leaves the page in the attached state, causing the former.

We observe that diagnosing this exception-handling failure is challenging due to three key factors:

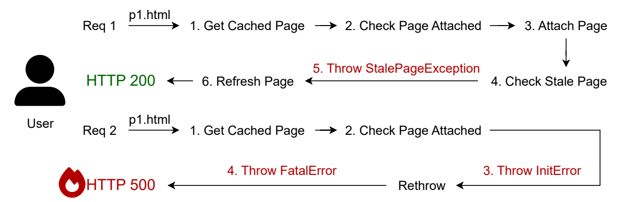

Figure 2: The request flow that triggers the HTTP 500 error when a user requests the same page twice. The labels correspond to the execution flow shown in Figure 1.

F1: Implicit state changes. Exceptions can have unexpected consequences beyond just modifying the control flow of a program. In the case of the `FatalError` exception shown in Figure 2, the presence of a `StalePageException` implicitly modifies the state of the `page` object. Specifically, the `detach` method is not called, leaving the object in an un-detached state that later triggers the `FatalError` exception. Importantly, there is no direct control-flow relationship between the `StalePageException` and the `FatalError`, meaning that simply tracing the execution path of the second request does not reveal the root cause of the failure.

F2: Silent exception handling. The root cause exception `StalePageException` is silently swallowed and not logged (Line 15). This allows the first request to continue normally despite encountering errors, improving overall system reliability and user experience. Such silent exception handling is a common practice among practitioners[59]; however, the fact that the exceptions are not logged or reported to external systems makes it difficult to diagnose failures that may occur later on (e.g., in the second request in this example).

F3: Spatially and temporally distant root cause. The error only surfaces in the second request, which can be temporally distant from when the root cause was triggered by the first request. Many unrelated error messages appear between the temporally-distant root cause and failure, and identifying if an error message is causally related to the failure can be challenging.

These key factors are not specific to the motivating example alone. For example, in a study of 10 Java libraries, Fetzer et al. found that 40% of exceptions caused implicit state changes [23], where both the control- and data-flow of the program are changed. Such state changes can cause implicit EDFs that are difficult to diagnosis. Moreover, an empirical study by Fu and Ryder found that approximately 40% of the exceptions caught by the analyzed applications were completely ignored by the program [25]. Existing empirical evidence shows the ubiquitous of the above factors in diagnosing EDF. Our experiments in §6 sampled 11 reproducible EDFs from ten popular apps (Table 2); six of the EDFs turned out to involve multiple requests/operations, and the root cause exceptions were missing in the logs for five EDFs.

2.2 Prior Work and Limitations

We now briefly discuss why closely related work is not applicable in identifying the root cause of an EDF.

Failure diagnosis. Statistical debugging [39] and backward slicing are two classic approaches to failure diagnosis. The former requires comparing many successful runs and failure runs to identify the failure root cause, a very different usage scenario from EXCHAIN. In our evaluation, even when provided with both types of runs, the leading statistical debugging tool, GZolta, detected 6 fewer root causes than EXCHAIN.

Backward slicing has struggled to balance accuracy and run-time overhead: static slicing [44, 49] cannot scale to analyze large-scale systems precisely. On the other, dynamic slicing techniques, like [5], can drastically slow down program execution—by up to 15 times in our evaluation, rendering them unsuitable for consistent use during routine testing or canary deployments.

Additionally, Sinha et al. suggested an integration of exception semantics with backward tracing, specifically to address null pointer exceptions. However, this technique's scope remains limited as it primarily tracks NULL propagation, making it non-generalizable for other exceptions [49].

Failure monitoring. Existing failure monitoring techniques primarily focus on identifying the failure of distributed systems [4, 13, 14, 17, 22, 26, 27, 28, 37, 38, 54]. However, identifying the failure service does not reveal the root cause of the failure. Panorama [31] and OmegaGen [40] improve the observability of large systems by monitoring grey-failures [30]. Such techniques are not sufficient to identify the root cause for EDFs as not all exceptions are triggered by grey failures; i.e, they do not work for silent exception handling (F2).

Log enhancement and analysis. These techniques focus on improving the log quality [58, 63], like logging more variable values at more selected program locations, and identifying failure-related logs [19] during post-mortem analysis. They are orthogonal to EXCHAIN. EXCHAIN conducts its analysis at run time, without relying on logs. Furthermore, no matter how many variables are logged at how many program locations, exception dependency cannot be figured out without the dependency analysis that we will present later. Notably, in our evaluations, only 3 failures benefitted from analyzing the first exception thrown by the application and the closest exception to the final failure.

2.3 Our Goal

Our goal is to build a tool that can automatically identify, at run time, the causal relationship between root cause exceptions and an EDF, even when the root causes are far from the failure, exceptions are silently swallowed, and dependencies are implicit. More precisely, given an EDF e_f, we aim to produce a DAG such that (1) there is a single sink node e_f, (2) source nodes represent root cause exceptions, and (3) an edge $e_i \rightarrow e_j$ indicates that e_j implicitly or explicitly depends on e_i. In the most common case, the output is a chain of exceptions, starting from the root cause exception and ending at e_f. For example, for the aforementioned Wicket failure in Figure 1, we produce the chain `StalePageException@32` $\rightarrow$ `InitError@9` $\rightarrow$ `FatalError@19` (while this notation only shows line numbers, the actual dependencies are between the *objects* corresponding to the exceptions thrown at run-time). Such dependencies can better explain to developers how root cause exceptions lead to the failure.

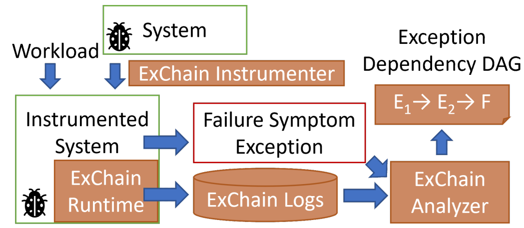

Figure 3: A high-level overview of EXCHAIN.

We aim to make our tool *easy to use*: a developer should be able to use the tool with low manual effort. The tool should be *accurate*: it should be able to find root causes of most EDFs, without generating many false positives. Finally, it should be *efficient*: its run time overhead would be modest. We envision EXCHAIN being deployed in a test, a canary, or a reproduction environment where a modest run time overhead is acceptable for the advantage of an easy-to-use and effective failure diagnosis tool.

3 EXCHAIN Overview

We begin with a high level overview of its workflow and how we envision EXCHAIN being used before diving into the technical challenges.

3.1 A High Level View

EXCHAIN consists of three components as shown in Figure 3: the *instrumenter*, the *runtime*, and the *analyzer*.

To use EXCHAIN, a developer *proactively* uses the automated instrumenter to instrument the target application binaries. The instrumenter does not require application source code or any application-specific configuration. The instrumented binaries are then deployed in the same way as the original binaries (e.g., in an integration testing environment or in a canary deployment).

As the instrumented application executes, EXCHAIN's *runtime* intercepts all exceptions that are raised and saves all exceptions as well as the critical runtime information required to determine their dependencies in the EXCHAIN log file. After a failure, the developer uses EXCHAIN *analyzer* to diagnose the failure. We assume that the developer charged with this incident troubleshooting and remediation knows the final symptom exception of the failure (potentially from the application's own log file). For instance, in our example from earlier, the developer knows that the HTTP 500 error occurred with the `FatalError` exception inside the `Page-`

`Provider` class. The analyzer takes this symptom exception as an input, and uses ExChain logs to output an exception DAG (or, most commonly, an exception chain) that has the given symptom exception as the sink. For instance, applying ExChain to the incident from Section 2, it produces `StalePageException@32` → `InitError@9` → `FatalError@19`. The source nodes in the analyzer output represent potential root cause exceptions.

3.2 Technical Challenges

ExChain proactively monitors all exceptions and collects runtime information every time an exception is thrown, treating every exception as a potential threat to the system. This is because predicting whether an exception will eventually cause a failure is impossible in general, and hence a selective interception may miss exceptions that are causally related to the failure. Similarly, a reactive strategy can miss important information if an exception and its information is not available when the failure happens.

The key challenge ExChain addresses is determining possible dependencies between two intercepted exceptions. There are many existing solutions to track explicit dependencies across exceptions, i.e., when one exception is thrown from the `catch` block of another. In this case, simply logging both the caught and the thrown exceptions can trivially capture their explicit dependency. There also exist program analysis tools to automatically construct such explicit dependencies across exceptions [24, 25, 33, 48]. However, these solutions cannot identify *implicit dependencies* of two exceptions where one exception causes state changes, which later causes the second exception.

The core contribution of ExChain is the ability to infer implicit dependencies of exceptions by tracking how an exception changes application states and how the changes cause subsequent exceptions. ExChain needs to address two challenges to achieve this.

First, ExChain needs to identify a set of application states to track. The set should be minimal in order to reduce the tracking overhead. To this end, ExChain uses two novel program analysis techniques to identify a small set of program memory locations to track. In particular, for each exception e_i, it identifies a set A_{e_i} of *affected memory locations* whose values are impacted by e_i and a set R_{e_i} of *responsible memory locations* whose values may cause e_i. For example, if an exception e_1 causes `null` values of the variables v_1 and v_2, and accessing v_1 later leads to a null pointer exception e_2, then $A_{e_1} = \{v_1, v_2\}$ and $R_{e_2} = \{v_1\}$. The fact that A_{e_1} and R_{e_2} overlaps readily implies that e_2 (implicitly) depends on e_1. Note that our abstraction of affected and responsible memory locations captures explicit dependencies as well: if e_2 is thrown in the catch block of e_1, A_{e_1} and R_{e_2} both include e_1 and hence e_2 depends on e_1.

Second, an exception e_2 may depend on another exception e_1 only *indirectly*. For instance, suppose e_1 causes $v_1 = -1$, which causes $v_2 = -1$ (e.g., via the copy `v2 = v1`), which causes the array index out of bounds exception e_2 (e.g., when executing `arr[v2]`). In this case, e_1's affected memory locations $\{v_1\}$ do not overlap with e_2's responsible memory locations $\{v_2, arr\}$, rather they are related to each other through data- and control-flow. Taint tracking can accurately capture such indirect dependencies or variables; however, it can be prohibitively expensive (up to 50× overhead for some applications in our evaluation in §6). ExChain uses a novel technique that combines static and dynamic taint tracking of a subset of memory locations that is significantly lightweight compared to dynamic taint tracking (although it can miss a small fraction of dependencies). Given the affected memory locations A_{e_1} of exception e_1, the techniques computes $Prop(A_{e_1})$, the set of all memory locations that are tainted by A_{e_1}. Using the information, ExChain decides that an exception e_2 with responsible memory location R_{e_2} depends on e_1 if the intersection of $Prop(A_{e_1})$ and R_{e_2} is not empty. Intuitively, a nonempty intersection means e_1 affects at least one memory location that, through data- and control-flow, affects at least one memory location that is responsible for e_2, and hence e_2 depends on e_1.

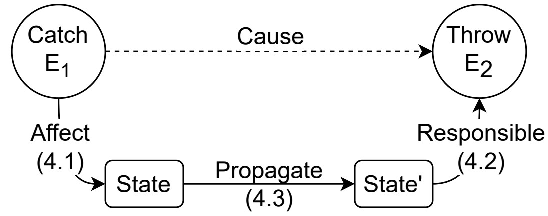

Figure 4: ExChain identifies the affected state of each exception and tracks its propagation. A causal link is established if the state causes another exception directly or indirectly.

3.3 Scope and Limitations

ExChain has several sources of false negatives. First, ExChain cannot identify affected and responsible memory locations that are not initialized when the exception is thrown. Second, ExChain cannot track the state propagation if the exception is thrown or caught by native code or if the state propagates to other systems (e.g. an exception causes a corrupted file or disrupts API functionalities).

The ExChain is specifically tailored for programming languages that utilize exceptions for error handling, such as C# and Java. Its algorithms analyze throw and try/catch statements to track where exceptions are raised and handled. However, ExChain is less effective for languages like C and Rust, which predominantly use return values for error handling, without raising and handling failures explicitly.

4 Detailed Design

Next, we describe the detailed design of ExChain to realize the workflow from the previous section. We start by describ-

Statement	Variable	Memory Location
`StalePageException:resolve`		
`detach();`	`this`	`Page`
`StalePageException:processRequest`		
`counter++;`	`counter`	`PageProvider.counter`
`page.refresh();`	`page`	`Page`

Table 1: Affected state analysis for the Wicket example.

ing the main analyses in ExChain (ref. Figure 4). Note that all analyses are *dynamic* unless specified otherwise, and they track *memory locations*, which are either local variables on the stack, objects in the heap, or fields of objects in the heap.

4.1 Affected State Analysis

First, given an exception, our goal is to identify all memory locations whose *values may be affected* by the exception, i.e., their values will differ depending on whether the exception is thrown or not. For example, in Figure 1, let us consider the control-flow of the program if the `resolve` method does not throw the `StalePageException@32`. Firstly, the `resolve` method calls the `detach` method (Line 11), which modifies the internal state of the `Page` object. Next, the `processRequest` method increments the `counter` variable (Line 12). Finally, the `page.refresh()` call (Line 17) is not executed because it is inside the catch block. Therefore, the `StalePageException@32` affects the memory locations corresponding to the `Page` object referenced by `this` variable at line 35, the `counter` field, and the `Page` object referenced by `page` variable at line 17.

However, obtaining this information at runtime can be challenging. Simply removing the `throw` statement and rerunning the request may not yield the correct result, particularly if the system is stateful.

To enable ExChain to identify memory locations affected by an exception at run-time, we develop a novel a static dataflow analysis that resembles liveness analysis [47]. Given an exception e and its corresponding stack trace ST from the current execution as input, ExChain generates a set of memory locations whose value will be altered by the exception. We represent a stack trace as a sequence of $k \geq 1$ tuples $\langle method, loc, vars \rangle$ corresponding to stack frames when the exception was thrown, where loc is the program location of the call site (for the first $k-1$ frames) or the `throw` statement (for the k-th frame), and $vars$ is a mapping of variables to their values. Our algorithm for computing the affected locations A is as follows:

1. Add thrown exception e to A.
2. For each stack frame $\langle method, loc, vars \rangle \in ST$:
   (a) Identify all instructions I_{aff} that are control dependent on the throw instruction or the corresponding invocation site loc.
   (b) For each instruction $i \in I_{aff}$ which is of the form $x = y$ or $x.f = y$ or $x.foo()$, determine (respectively) the assigned local variable, the assigned object field, or the object on which a method was invoked, and add these locations to A. Intuitively, this is because their value may be impacted by the change in control flow due to the exception. Note that concrete memory locations are obtained by resolving object references and fields via $vars$.
3. Return all affected locations A.

Table 1 shows the analysis result of the `StalePageException`. When the `StalePageException@32` is thrown, the `detach()` statement is control dependent of the `throw` statement. The `detach()` statement is a method invocation of the object referenced by `this`. Therefore, ExChain identifies the memory location pointed by `this` as affected. The stack trace of this exception also contains the method `processRequest` which is at the invocation site `page.resolve(req.sessionId)` (Line 11); as this call is aborted, the control-dependent statements `counter++` and `page.refresh()` are marked as affected. Correspondingly, the affected state analysis returns two memory locations: (1) the `Page` object referenced by `this` in the `resolve` method and by `page` in the `processRequest` method, and (2) the class field `counter`.

4.2 Responsible State Analysis

Given an exception, we also need to identify all memory locations whose specific values *can cause* the exception. Note that exceptions broadly have two types of causes: (a) exceptions originating at a `throw` statement are usually caused by some program condition that is checked by an enclosing `if`; and (b) run-time exceptions can be triggered while executing expressions because of the value in some memory location (e.g., if a reference is `null` or if a divisor is zero).

As an example, consider the `processRequest` method shown in Figure 1. This method can throw an `ArrayIndexOutOfBoundsException` at Line 5 if `cache.length <= req.pageIndex`. The memory location referenced by `cache` and `req` are responsible for the exception. Next if `page` is attached, the method throws an `InitError` exception (Line 9). In this case, the `Page` object referenced by `page` is the responsible memory location, since it is part of the closest enclosing `if` condition. Note that the `InitError` is thrown indirectly by a wrapper method called `logAndThrow`. Therefore, simply analyzing the method that directly throws the exception is not sufficient to identify the responsible location. To address this, we implement several heuristics based on the semantics of the exceptions and the structure of the code.

Exception rethrown. Many exceptions are thrown explicitly in the catch block (e.g. the FatalError in Figure 1). The memory location referenced by the caught exception (e.g. `e2` at Line 18) is responsible for the new exception.

Run-time exceptions without an explicit throw statement. We maintain a list of exceptions that are thrown directly by the runtime while executing individual instructions, and handle them specially to identify the memory locations responsible for these exceptions. For instance, if a `NullPointerException` is triggered by a method invocation instruction or an object field access, then EXCHAIN identifies the corresponding object reference as the culprit.

Exceptions thrown by throw statements. When an exception is thrown by a `throw` statement, EXCHAIN employs a backward control-flow analysis to identify the memory locations responsible for the exception. The algorithm takes an exception e and its corresponding stack trace ST as input and returns a set of memory locations R that are responsible for the input exception. The stack trace is again represented as a sequence of tuples $\langle method, loc, vars \rangle$ as before. The algorithm is as follows:

1. For each frame $\langle method, loc, vars \rangle \in ST$ starting from the top of the stack:
   (a) Find the closest branch statement of the current frame location *loc* based on the control-flow graph of the method.
   (b) If no such branch is identified in this method, go to step 1.
   (c) Collect all variables or object fields that are referenced by the condition expression of the branch statement. Use *vars* to resolve variables to memory locations, and add these to R. Break and go to step 2.
2. Return the set R.

For example, in the `InitError` shown in Figure 1, EXCHAIN first analyzes the `logAndThrow` method at line 24, and then the `processRequest` method at line 9. Since the `throw` statement in question is not dominated by any branch condition in the `logAndThrow` method, the analysis continues up the call stack to the `processRequest` method, where it identifies the closest branch condition as `page.isAttached()`. EXCHAIN then resolves the local variable reference and returns a singelton set containing the memory location referenced by `value` as R. Since this value is the same object included in the affected set for `StalePageException` (ref. Section 4.1), EXCHAIN can establish causality between `StalePageException` and `InitError`.

4.3 Hybrid Taint Flow Analysis

In general, EXCHAIN needs to consider how the values affected by some exception e_i propagate to other values before they become responsible for some other exception e_j (§3.2). This is done using *taint analysis*. The main idea behind taint analysis is to associate some information with program values (e.g., that they are affected by exception e_i) and propagate this to other values that are derived from the former.

Traditional *dynamic* taint analysis works by instrumenting program code to propagate taint information at every instruction, such as copying local variables, performing arithmetic computation, or invoking method calls. This instrumentation introduces excessive overhead to the application [11, 15, 20],

```
1  class Foo {
2      int value = 0;
3      Taint valueT = new Taint("const:0");
4      Taint thisT = new Taint("obj:Foo");
5  }
6  void m() {
7      int i1 = 10;
8      // create a new taint because i1 is created
       ↪  from a constant.
9      Taint i1T = new Taint("const:10");
10     int i2 = i1;
11     // Passing the taint information from i1 to
       ↪  i2.
12     Taint i2T = i1T;
13     Foo e1 = new Foo();
14     Foo e2 = e1;
15     int i3 = e1.value;
16     // Passing the taint information from
       ↪  e1.value to i3.
17     Taint i3T = e1.valueT;
18 }
```

Passing the taint information
from e1 to e2 is not necessary!

Figure 5: A simple program to demonstrate how dynamic taint analysis tools tracks the taint tag for heap objects and local variables. The original code is not highlighted.

making it difficult to apply dynamic taint analysis techniques to large, complex enterprise-level applications even in an integration/canary test environment. Alternatively, *static* taint analysis reconstructs the dynamic behavior of a program using only static code analysis [8, 41], which introduces zero overhead to the application while trading off precision.

Hybrid taint analysis. Neither static nor dynamic taint analysis alone can achieve both accurate and efficient taint tracking. Therefore, it is natural to ask if it is possible to combine these two approaches to achieve a high precision and a low overhead. EXCHAIN implements a novel hybrid taint analysis that leverages the following two observations.

First, the main overhead introduced by dynamic taint analysis comes from maintaining the taint information for local primitives, such as integers and booleans. In contrast, tracking heap objects can be done more efficiently and accurately by adding only taint information to the heap object itself.

For example, Figure 5 illustrates an instrumented Java program that tracks taint information using dynamic taint analysis. The fields `valueT` and `thisT` of the class `Foo` are used to track the taint information of `Foo.value` and `Foo` respectively. The dynamic taint analysis tool creates a taint reference for each local primitive and updates the taint information accordingly (Line 9, 12, and 17). For object references, the tool does not create taint references if they point to heap objects whose taint information is already being maintained by corresponding fields (Line 3-4). When the `thisT` field of an object is updated, all references pointing to that object are automatically updated as well. In our evaluation, tracking local primitives introduces 87-5005% overhead but tracking heap objects only introduces 1-10% to the system.

Second, local primitives can be efficiently tracked using

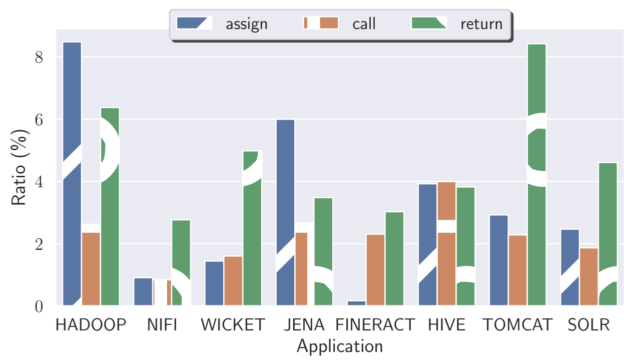

Figure 6: Analysis result of the percentage of data-flows between heap and local objects using CodeQL.

static taint analysis offline and static taint analysis produces more accurate result for local primitives compared to heap objects [29]. Heap objects can be manipulated in various ways by the program, such as being passed between functions or being dynamically allocated and deallocated. This makes it harder to track the flow of data through the program and to accurately determine which inputs have tainted a particular object. On the other hand, local primitives are typically only modified within a single function or block of code, making it easier to trace their flow of data.

ExChain utilizes these observations and dynamically instruments heap objects and adds taint information at run time, introducing only a constant overhead per exception. For local primitives, ExChain uses static taint analysis to track data-flow offline, introducing no overhead at run time.

One limitation of tracking different types of variables differently is that it may miss data flow between heap objects and local primitives. For example, in Figure 5, taint information will be lost between `e1.value` and `i3` because ExChain maintains the taint information of `e1.value` dynamically and the taint information of `i3` statically. This limitation may affect the accuracy of ExChain's analysis.

To better understand the impact, we used CodeQL [16] to statically analyze multiple popular cloud services. Specifically, we focused on three types of statements: assignments, method calls, and method returns. Figure 6 presents the results of our analysis. Across all applications, we found that less than 8.5% of assignment statements, less than 4% of method call statements, and less than 8.4% of method return statements had data flow from heap objects to local primitives. Our findings confirmed that the majority of data flow occurs between heap-to-heap and local-to-local, which can be effectively handled by hybrid taint analysis. In our evaluation of 11 reproduced failures, we found that ExChain can identify the root cause of 6 issues by only tracking heap objects, while four issues require only tracking local primitives. Only one failure requires tracing the data-flow between heap objects and local primitives. Besides, tracking local primitives introduces 17x overhead to the target system compared to only tracking heap objects. Overall, by selectively tracing heap objects and local primitives, ExChain can identify the root cause of most failures with minimal overhead.

4.4 Putting it Together

We now describe how different pieces fit together in the workflow of ExChain as shown in Figure 3. At runtime, ExChain intercepts every exception that is thrown by the application. On each exception, ExChain performs both affected and responsible state analysis and logs the results in the ExChain logs (the results are cached and reused when the same exception is thrown multiple times). It also performs the dynamic part of its hybrid taint analysis. For affected states that are heap objects, ExChain marks them with a unique ID of the exception. For affected states that are local primitives, ExChain logs the stack slot number and the corresponding exception. When analyzing responsible states, ExChain checks if the memory location is a heap object that contains any labels of previous exceptions. If a label is found, ExChain records in its logs the causal link between the current exception and the exception represented by the label. If the memory location is a local primitive, ExChain logs the stack slot number and the corresponding exception.

To diagnose a failure, the user uses the ExChain analyzer offline with the target symptom exception of the failure. The analyzer first performs the static part of its hybrid taint analysis with the stack slots of affected states as sources and the stack slots of responsible states (the states are retrieved from ExChain logs). If the static taint analysis reports a flow from a source to a sink, ExChain reports the causal link between the corresponding exceptions. Finally, the analyzer returns a DAG (or most commonly, a chain) that has the symptom exception as the sink node. The source nodes in the returned DAG represents the root cause exceptions.

5 Implementation

Our ExChain implementation consists of 9,000 lines of code written in Kotlin, Java, and C++. It instruments the compiled bytecode of the target system and attaches dynamic monitors to the runtime. We use JVM Tool Interface (JVMTi) [35] to capture all exceptions thrown by the target application and map local variables and heap fields to memory locations. The affected and responsible state analyzer are developed on top of ASM [9]. Our dynamic taint analysis is based on Phosphor [11], and the static taint analysis is implemented using Soot [53] and FlowDroid [8]. Although our algorithm is not tied to a specific implementation of JDK and JVM, we only execute target applications using OpenJDK 16 because the underlying dynamic taint analysis tool requires APIs that are only available after OpenJDK 16.

Dynamic Entry Point Inference. We have also implemented a dynamic entry point inferring technique that allows the static taint analysis to provide more accurate results.

Client-server architecture for cloud services typically involves multiple public interfaces, each serving as an entry

```
int main() {
    while (true) {
        Request r = waitForNewRequest();
        dispatchRequest(r);
    }
}
class Server {
    @Endpoint("create")
    Response createUser(Request r) {
        method1(); // may throw Exception1
    }
    @Endpoint("remove")
    Response removeUser(Request r) {
        method2(); // may throw Exception2
    }
}
```

Figure 7: A simple web server with two endpoints.

```
Exception1 ->
  Server.method1,
  Server.createUser,
  Thread.run
Exception2 ->
  Server.method2:24,
  Server.removeUser,
  Thread.run
Exception1 ->
  Server.method1,
  Server.createUser,
  Thread.run
```

```
void main() {
    createUser(SYM);
    removeUser(SYM);
    createUser(SYM);
}
```

Figure 8: ExChain constructs a main method based on the exception trace collected at runtime.

point to the application. Figure 7 illustrates a simple web server with two public interfaces: `create` and `remove`. The main function of the server is a hot loop that waits for incoming requests and dispatches them to respective endpoints.

The dynamic design of modern web frameworks such as Spring [51] and Wicket [7], which dispatch requests to different worker threads through reflection and dependency injection, makes it difficult for the static taint analyzer to construct an accurate call graph [6]. This design significantly limits the completeness of the static analysis.

To address this problem, ExChain leverages the exception traces collected at runtime. For each exception, ExChain identifies the deepest stack frame that contains application code based on the package name of the caller. It then constructs a main method that invokes the corresponding application code in sequential order. For example, Figure 8 demonstrates an exception trace of Figure 7. ExChain finds the deepest method that contains application code for each exception and creates a new main method that invokes the identified methods sequentially. By doing so, ExChain is able to generate a new main method for the application that starts with the identified application code.

In our evaluation, dynamic entry point inference helps ExChain to identify the root cause for two more issues compared to using the original main method of the application.

Issue	Multi-Run	Cause Logged	# Excp Dist	# Excp Total
WICKET-6908	✓	✓	5	8
JENA-324	✓	✓	792	796
FINERACT-1211	✗	✓	1	58
MAPREDUCE-6654	✗	✓	11	117
HADOOP-17812	✗	✗	1	24
WICKET-6249	✓	✗	7	11
HDFS-4128	✓	✗	7	115
HIVE-13410	✓	✓	15	51
NIFI-8249	✓	✓	1	47
SOLR-16363	✗	✗	1	171
TOMCAT-65131	✗	✗	1	13

Table 2: Basic information of EDFs. 'Multi-Run' indicates if the root cause and the final exception occur in different operations. 'Cause Logged' shows if the root cause exception is logged by the application. 'Dist' shows the number of exceptions thrown between the root cause and the final exception. 'Total' shows the total number of exceptions thrown by the application during the reproduction.

6 Evaluation

We evaluate ExChain to answer the following questions: (1) How does ExChain compare to state-of-the-art failure diagnosis techniques in identifying the root cause for EDFs? (2) How do our analysis techniques help improve the accuracy-performance tradeoffs of ExChain? We conduct all experiments on an Ubuntu server with an Intel Xeon 1290P processor and 128 GB of memory. We set an 8-hour time limit for static taint analysis, with a maximum heap size of 32 GB.

6.1 Methodology

For our evaluation, we looked at popular open source applications maintained by the Apache Foundation. We query the Jira issue tracking system [32] with a list of keywords such as "exception handling". From the result of our query (run on 11-15-2022), we closely examined the latest 30 issues return that are indeed software bugs related to exception handling and have clearly described failure symptoms, and were able to reproduce 11 of them (the remaining ones do not provide an instruction to reproduce the failure).

As such, these 11 *reproducible* incidents spanning 10 open source applications form the core of our evaluation setup. Table 2 shows the basic information of the reproduced 11 issues. We reproduce all failures by running the service in production configuration except for NIFI-8249, which uses a customized class loading mechanism that is not supported by the underlying taint analysis framework. As a result, we reproduced this issue using unit tests. We manually analyze the issue report summary and developer conversations in the ticket to identify the root cause to serve as the ground truth. Note that our system does not need or have access to this issue report.

As shown in Table 2, many of the failures are non-trivial:

Issue	ExChain	Statistical		SL4J	Log	
		Ochiai	Taran.		First	Nearst
WI-6908	✓	✗	✗	N/A	✗	✗
JE-324	✓	✗	✗	✗	✓	✓
FI-1211	✓	N/A	N/A	N/A	✗	✗
MA-6654	✓	✗	✗	N/A	✗	✗
HA-17812	✓	✗	✓	N/A	✗	✗
WI-6249	✓	✗	✗	N/A	✗	✗
HD-4128	✓	✓	✓	N/A	✗	✗
HI-13410	✓	✗	✗	N/A	✓	✗
NI-8249	✗	✗	✗	N/A	✓	✓
SO-16363	✗	N/A	N/A	N/A	✗	✗
TO-65131	✗	N/A	N/A	N/A	✗	✗

Table 3: The analysis result of each issue. Ochiai and Tarantula are two statistical debugging techniques implemented by GZoltar. N/A means GZoltar and Slicer4J are not applicable to the target application. First and Nearest are two debugging techniques that focus on examining the initial exception thrown by the application and the closest exception to the final failure from logs.

their root causes and failures happen in different executions, root causes are not logged, and many (unrelated) exceptions separate the failures from their the root causes.

6.2 End-to-End Evaluation

For the end-to-end evaluation of accuracy, we compare ExChain with three state-of-the-art fault localization techniques: statistical debugging (GZoltar [12] with two different ranking algorithms Ochiai [3] and Tarantula [34]), slicing (Slicer4J [5]) and Log analysis. GZoltar requires workloads that contains both pass and fail cases and we use the unit tests associate with the application. If the existing unit test does not cover the reproduced failure, we manually implement one. GZoltar return a ranked list of statements that may be related to the failure based on their relevance to the failure and Slicer4J returns a ranked list of statements that are data dependent of the failure. If the containing method of a statement reported by the tool that throws the root cause is ranked top 200 of the list, we report a true positive. In our analysis of application logs, we utilized two strategies: "First" representing the initial exception thrown and "Nearest" representing the closest exception to the final failure.

Note that, as discussed in Section 2.2, it is much more costly to use GZoltar and Slicer4J: GZoltar requires many successful runs and failure runs, and Slicer4J requires re-executing a failed run. In contrast, ExChain allows diagnosis right after a failure run.

Table 3 presents the analysis result. ExChain successfully identified the root cause for most issues (8 out of 11). GZoltar only identified the root cause for 2 issues among the 8 on which we could run it. GZoltar was unable to analyze Fineract, Tomcat, and Solr due to incompatibilities with the building system used by these applications. For HDFS-4128, Ochiai identified the root cause in the top 2 predicted statements. However, for HADOOP-17812 and WICKET-6249, Ochiai failed to predict the root cause within the top 200 statements. For Tarantula, it predicted the root cause statements within the top 30 statements for HADOOP-17812 and HDFS-4128. For 5 issues, GZoltar cannot report any root cause statements due to insufficient pass/fail executions. Our experimental results highlight that ExChain is more capable than GZoltar in identifying the root cause for EDFs.

To gain insight into the challenges of diagnosing failures using dynamic variable dependency tracking, we performed a backward slicing for all failures using Slicer4J [5]. However, we were able to use the tool to reproduce only one issue: JENA-324; Slicer4J could not used with the other issues due to incompatible Java versions. For JENA-324, Slicer4J reported 3741 statements related to the final failure, with the root cause identified at the 3628th statement. This suggests that relying on variable dependency for failure diagnosis can lead to information overload, potentially overwhelming developers. Slicer4J can introduce significant performance overhead (up to 15×) Ahmed et al. [5]).

Finally, for the log based approaches, the "First" strategy identified the root cause for three out of the total failures. The "Nearest" strategy pinpointed the root cause for only two distinct failures. It's crucial to note that even in our experiments with smaller workloads, numerous exceptions can occur before and after the root cause, particularly in long-running services.

6.3 Accuracy vs. Performance Tradeoff

To put the accuracy-performance tradeoff of ExChain in context and explain the value of our optimizations, we consider two other hypothetical designs SI+Static and SI+Dynamic (SI stands for affected/responsible state identification). Similar to ExChain, both log exceptions at runtime and identify affected and responsible states using the algorithms mentioned in §4.1 and §4.2. They differ in how they analyze taint flow between affected to responsible states: SI+Static uses fully static taint analysis and SI+Dynamic uses fully dynamic taint analysis. In contrast, ExChain uses a hybrid taint analysis: dynamic analysis of heap objects and static analysis of local primitives. Note, that SI+Static logs all exceptions as well as the corresponding stack traces and runs affected and responsible state analysis offline. SI+Static is expected to offer low run-time overhead but least accurate diagnosis results. Conversely, SI+Dynamic is expected to offer the highest overhead but also the highest diagnosis accuracy.

To evaluate the performance impact of ExChain, we identify benchmarks for 7 out of the 10 applications. For Hadoop, Solr, MapReduce, and HDFS, we used the built-in benchmarks to generate workloads [18, 45, 50, 52]. For Fineract, Wicket, and Tomcat, we use the Apache HTTP benchmarking tool [1] to measure their performance. For each benchmark, we measure both throughput and latency. We encountered issues when attempting to benchmark NIFI in production mode due to dynamic class loading as described earlier. We were

Issue	ExChain		SI+Static		SI+Dynamic	
	TP	FP	TP	FP	TP	FP
WICKET-6908	✓	1	✓	1	✓	0
JENA-324	✓	0	✗	0	✓	0
FINERACT-1211	✓	0	✓	0	✓	0
MAPREDUCE-6654	✓	0	✗	0	✓	0
HADOOP-17812	✓	0	✗	0	✓	0
WICKET-6249	✓	0	✓	0	✓	0
HDFS-4128	✓	0	✗	0	✓	0
HIVE-13410	✓	0	✗	0	✓	0
NIFI-8249	✗	0	✗	0	✓	0
SOLR-16363	✗	0	✗	0	✓	0
TOMCAT-65131	✗	0	✗	0	✓	0
Sum	8/11	1	3/11	1	11/11	0

Table 4: Analysis result of each issue. ExChain cannot identify the root cause for NIFI-8249 and SOLR-16363 because of the imprecise analysis result returned by the underlying static taint analysis tool. ExChain cannot identify the root cause for TOMCAT-65131 because of the data-flow between heap objects and local primitives.

also unable to find a representative benchmark workload for Jena and Hive.[2] Solr only reports throughput and MapReduce only reports latency. All remaining benchmarks report both latency and throughput.

Accuracy Results. Table 4 presents the accuracy results for ExChain and two baselines including the number of true positives (TP) and false positives (FP). A true positive for a technique means that it successfully identifies the root cause exception and correctly reported causal relationship between the root cause exception and the final failure described in the issue. A false positive means that the technique reports an exception that is not mentioned by the reporter and fixing the exception does not prevent the final failure.

The result shows that ExChain successfully identified the root cause for most issues (8 out of 11) with only 1 false positive. SI+Dynamic successfully identified all root causes without any false positive (at the cost of huge run-time overhead that we discuss later). On the other hand, SI+Static could identify root causes for only 3 issues with 1 false positive.

Recall that SI+Dynamic is based on the affected and responsible variables identified by our analysis described in §4. The fact that it can successfully identify all root causes shows the effectiveness of the analysis algorithms.

We also investigated the three failures for which ExChain failed to report the true root cause. For two of the cases (NIFI-8249 and SOLR-16363), ExChain failed because of the imprecise analysis result returned by the underlying static taint analysis while tracking local variables. Only for one case (TOMCAT-65131), ExChain failed because of its design limitation of not being able to track data-flow between heap and local objects.

Performance Result. Figure 9 shows the latency results reported by 6 applications. On average, ExChain incurred 1%-12% overhead on latency, only 2% more than SI+Static. In contrast, SI+Dynamic incurred 87%-5015% overhead. The throughput result in Figure 10 shows a similar trend: ExChain incurred 1%-11% degradation on throughput, while SI+Static also incurred 1%-11% degradation. In contrast, SI+Dynamic incurred 48%-99% degradation.

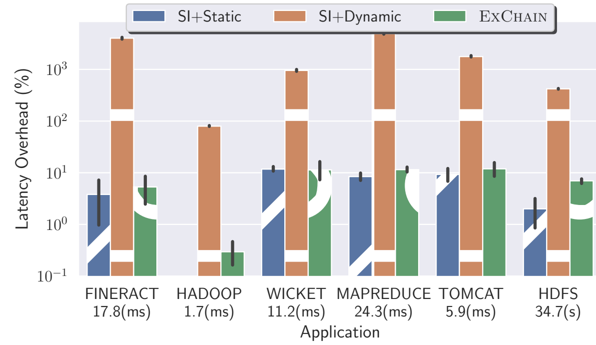

Figure 9: The latency overhead for different applications in *log scale*. Lower is better.

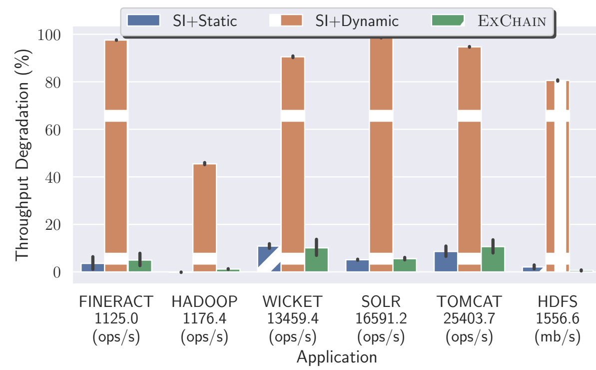

Figure 10: The throughput degradation for different applications. Lower is better.

Our evaluation shows that ExChain achieves a better balance between performance and accuracy than SI+Dynamic and SI+Static. Specifically, it achieves an accuracy closer to SI+Dynamic, with a cost closer to SI+Static. In fact, ExChain successfully identified the root cause of all failures whose affected and responsible states are heap objects, whereas SI+Static only identified the root cause for 3 such failures. Moreover, ExChain reported only one false positive out of 11 issues, demonstrating its high accuracy. In terms of performance, ExChain introduces an average overhead of only 8%, making it feasible to deploy in an integration test or canary environment.

7 Discussion

Our current focus was on using ExChain was in a test/canary environment where a moderate performance overhead ($\approx 8\%$) may be acceptable. One natural question is if our approach may eventually be amenable to be run in production with a lower overhead.

[2] A third-party benchmark for an old version of Hive was available, but our dynamic taint analysis tool could not instrument the benchmark application.

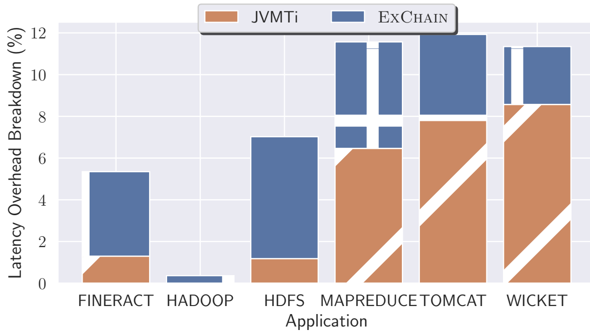

Figure 11: Latency overhead break down for different applications.

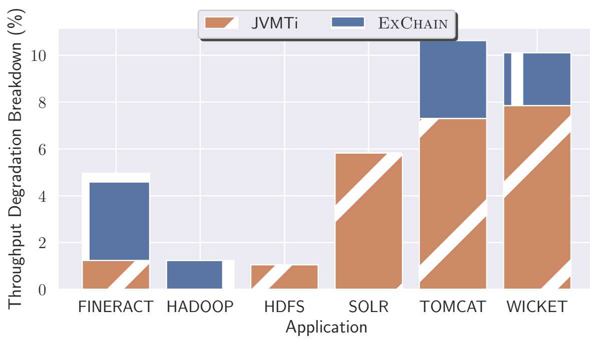

Figure 12: Throughput degradation break down for different applications.

To understand the future feasibility of extending ExChain we investigate the sources of the overhead. We divide the system into two components: JVMTi and ExChain. Figure 11 and Figure 12 show the breakdown of overhead introduced by each component. JVMTi represents the aforementioned overhead due to JVMTi. ExChain represents the overhead introduced by the central design, including logging all exceptions and their corresponding stack traces, computing the affected and responsible states, and storing taint information of heap objects.

From our observations, less than half of the total overhead is attributed to the core components of ExChain, while the rest originates from JVMTi. JVMTi can disable several JIT optimizations when attached to the JVM, which affects the overhead. An alternative way to intercepting exceptions (e.g., through instrumentation or a better JVM mechanism similar to .NET's first-chance-exception[21]) could reduce this overhead substantially to enable closer-to-production acceptable overhead ($\leq 5\%$).

8 Other Related Work

We discussed some key related efforts and their limitations in §2.2. Here, we discuss other related work.

Statistical debugging technique. There is a rich body of work focusing on statistical debugging [2, 12, 34, 43, 55, 57, 61]. Such techniques are effective if the developer provides both failing and passing executions. Unfortunately, such data is not always available. In our evaluation, we show that with existing test suit, GZoltar can only identify the root cause for one EDF. Moreover, statistical debugging aims to identify events (e.g., exceptions) that are *correlated* to failures, rather than finding the causal dependencies among multiple events.

Failure reproducing. Kasikci et al. showed that it is possible to reproduce failures with low overhead instrumentation using hardware features [36]. Pensieve reconstructs failing executions using dependency analysis of runtime events [62]. ExChain is complementary to failure reproducing techniques and help developers to pinpoint the root cause efficiently.

Failure handling testing. ChaosMachine [60] and Filibuster [42] use chaos engineering to test failure handling logic of the application. Such techniques are useful in identifying bugs in failure handling logic.

Speeding up dynamic taint analysis. JetStream uses parallel execution and record and replay techniques to improve the performance of dynamic information flow tracking [46]. Iodine uses static analysis to remote runtime monitors if the data-flow can be determined statically [10]. Both tool show that taint analysis is useful in debugging and failure diagnosis. ExChain uses exception-focused hybrid taint analysis and focuses on identifying the root cause for EDFs.

9 Conclusions

In some sense, ExChain solves a particularly hard problem — the very practices of good software engineering at scale (e.g., throwing exceptions, silent handling) also end up creating subtle exception-dependent failure modes that are incredibly hard to debug! Our key observation is that unlike basic exception chains, EDFs can entail subtle stateful dependencies between the root cause and the eventual failure mode.

In designing ExChain, we addressed fundamental challenges in applying program analysis techniques to balance the performance and overhead in tracking such stateful dependencies in exception handling failures. ExChain helps developers diagnose EDFs using a famililar exception-trace like abstraction akin to traditional debugging workflows. Our evaluation showed that ExChain is able to successfully diagnose subtle issues that stumped expert developers in popular applications with little to no manual effort and that it significantly outperforms state-of-art techniques. While our current implementation offers sufficient performance for test and canary deployments, our core design contributions are amenable to production deployments at scale as well.

10 Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and constructive feedback. This research was supported in part by seed funding from CMU's CyLab and by the NSF grant CCF-2119184. A portion of this work was carried out during an internship at Microsoft Research.

References

[1] ab. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed: 2023-02-23.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007. doi: 10.1109/TAIC.PART.2007.13.

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82:1780–1792, 2009.

[4] Marcos K. Aguilera and Michael Walfish. No time for asynchrony. In *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, HotOS'09, page 3, USA, 2009. USENIX Association.

[5] Khaled Ahmed, Mieszko Lis, and Julia Rubin. Slicer4J: A Dynamic Slicer for Java. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

[6] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. Static analysis of java enterprise applications: Frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 794–807, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386026. URL https://doi.org/10.1145/3385412.3386026.

[7] Apache Wicket. Apache wicket. https://wicket.apache.org/. Accessed: 2023-02-23.

[8] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259–269, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594299. URL https://doi.org/10.1145/2594291.2594299.

[9] asm. Asm: A java bytecode engineering library. https://asm.ow2.io/index.html. Accessed: 2023-02-23.

[10] Subarno Banerjee, David Devecsery, Peter M. Chen, and Satish Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 490–504, 2019. doi: 10.1109/SP.2019.00043.

[11] Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, page 83–101, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450325851. doi: 10.1145/2660193.2660212. URL https://doi.org/10.1145/2660193.2660212.

[12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: an eclipse plug-in for testing and debugging. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 378–381, 2012. doi: 10.1145/2351676.2351752.

[13] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, mar 1996. ISSN 0004-5411. doi: 10.1145/226643.226647. URL https://doi.org/10.1145/226643.226647.

[14] Wei Chen, S. Toueg, and M.K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(1):13–32, 2002. doi: 10.1109/12.980014.

[15] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 196–206, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937346. doi: 10.1145/1273463.1273490. URL https://doi.org/10.1145/1273463.1273490.

[16] Codeql. Codeql. https://codeql.github.com/. Accessed: 2023-02-23.

[17] A. Das, I. Gupta, and A. Motivala. Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proceedings International Conference on Dependable Systems and Networks*, pages 303–312, 2002. doi: 10.1109/DSN.2002.1028914.

[18] dfsio. Hadoop hdfs dfsio. https://github.com/c9n/hadoop/blob/master/hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-mapreduce-client-jobclient/src/test/java/org/apache/hadoop/fs/TestDFSIO.java. Accessed: 2023-02-23.

[19] Pradeep Dogga, Karthik Narasimhan, Anirudh Sivaraman, Shiv Kumar Saini, George Varghese, and Ravi Netravali. Revelio: Ml-generated debugging queries for finding root causes in distributed systems. In *Conference on Machine Learning and Systems*, 2022.

[20] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2), jun 2014. ISSN 0734-2071. doi: 10.1145/2619091. URL https://doi.org/10.1145/2619091.

[21] fce. Appdomain.firstchanceexception event. https://learn.microsoft.com/en-us/dotnet/api/system.appdomain.firstchanceexception?view=net-8.0. Accessed: 2023-02-23.

[22] C. Fetzer. Perfect failure detection in timed asynchronous systems. *IEEE Transactions on Computers*, 52(2):99–112, 2003. doi: 10.1109/TC.2003.1176979.

[23] C. Fetzer, K. Hogstedt, and P. Felber. Automatic detection and masking of non-atomic exception handling. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 445–454, 2003. doi: 10.1109/DSN.2003.1209955.

[24] C. Fu, A. Milanova, B.G. Ryder, and D.G. Wonnacott. Robustness testing of java server applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, 2005. doi: 10.1109/TSE.2005.51.

[25] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *29th International Conference on Software Engineering (ICSE'07)*, pages 230–239, 2007. doi: 10.1109/ICSE.2007.35.

[26] Trinabh Gupta, Joshua B. Leners, Marcos K. Aguilera, and Michael Walfish. Improving availability in distributed systems with failure informers. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 427–441, Lombard, IL, April 2013. USENIX Association. ISBN 978-1-931971-00-3. URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/leners.

[27] Andreas Haeberlen and Petr Kuznetsov. The fault detection problem. In Tarek Abdelzaher, Michel Raynal, and Nicola Santoro, editors, *Principles of Distributed Systems*, pages 99–114, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-10877-8.

[28] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 175–188, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935915. doi: 10.1145/1294261.1294279. URL https://doi.org/10.1145/1294261.1294279.

[29] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581134134. doi: 10.1145/379605.379665. URL https://doi.org/10.1145/379605.379665.

[30] 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 '17, page 150–155, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350686. doi: 10.1145/3102980.3103005. URL https://doi.org/10.1145/3102980.3103005.

[31] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 1–16, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/huang.

[32] jira. Asf jira. https://issues.apache.org/jira/secure/Dashboard.jspa. Accessed: 2023-02-23.

[33] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for java. *J. Syst. Softw.*, 72(1):59–69, jun 2004. ISSN 0164-1212. doi: 10.1016/S0164-1212(03)00057-8. URL https://doi.org/10.1016/S0164-1212(03)00057-8.

[34] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 467–477, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113472X. doi: 10.1145/581339.581397. URL https://doi.org/10.1145/581339.581397.

[35] JVMTI. Jvm(tm) tool interface. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html. Accessed: 2023-02-23.

[36] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 344–360, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450338349. doi: 10.1145/2815400.2815412. URL https://doi.org/10.1145/2815400.2815412.

[37] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 279–294, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450309776. doi: 10.1145/2043556.2043583. URL https://doi.org/10.1145/2043556.2043583.

[38] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming uncertainty in distributed systems with help from the network. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450332385. doi: 10.1145/2741948.2741976. URL https://doi.org/10.1145/2741948.2741976.

[39] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on software engineering*, 32(10):831–848, 2006.

[40] 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. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/lou.

[41] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, page 229–240, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316514. doi: 10.1145/2382196.2382223. URL https://doi.org/10.1145/2382196.2382223.

[42] 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. ISBN 9781450386388. doi: 10.1145/3472883.3487005. URL https://doi.org/10.1145/3472883.3487005.

[43] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3), aug 2011. ISSN 1049-331X. doi: 10.1145/2000791.2000795. URL https://doi.org/10.1145/2000791.2000795.

[44] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *2009 IEEE 31st International Conference on Software Engineering*, pages 133–143, 2009. doi: 10.1109/ICSE.2009.5070515.

[45] nnbench. Hadoop benchmarking. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/Benchmarking.html. Accessed: 2023-02-23.

[46] Andrew Quinn, David Devecsery, Peter M. Chen, and Jason Flinn. JetStream: Cluster-Scale parallelization of information flow queries. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 451–466, Savannah, GA, November 2016. USENIX Association. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/quinn.

[47] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.

[48] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, apr 2003. ISSN 1049-331X. doi: 10.1145/941566.941569. URL https://doi.org/10.1145/941566.941569.

[49] Saurabh Sinha, Hina Shah, Carsten Görg, Shujuan Jiang, Mijung Kim, and Mary Jean Harrold. Fault localization and repair for java runtime exceptions. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 153–164, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583389. doi: 10.1145/1572272.1572291. URL https://doi.org/10.1145/1572272.1572291.

[50] solr. Solr jmh-benchmarks module. https://github.com/apache/solr/tree/main/solr/benchmark. Accessed: 2023-02-23.

[51] spring. Spring | home. https://spring.io/. Accessed: 2023-02-23.

[52] terasort. Package org.apache.hadoop.examples.terasort. https://hadoop.apache.org/docs/r3.0.0/api/org/apache/hadoop/examples/terasort/package-summary.html. Accessed: 2023-02-23.

[53] 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.

[54] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In Nigel Davies, Seitz Jochen, and Kerry Raymond, editors, *Middleware'98*, pages 55–70, London, 1998. Springer London. ISBN 978-1-4471-1283-9.

[55] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374, 2009. doi: 10.1109/ICSE.2009.5070536.

[56] WICKET-6908. Wicket-6908. https://issues.apache.org/jira/browse/WICKET-6908. Accessed: 2023-02-23.

[57] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1): 290–308, 2014. doi: 10.1109/TR.2013.2285319.

[58] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, page 293–306, USA, 2012. USENIX Association. ISBN 9781931971966.

[59] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 249–265, USA, 2014. USENIX Association. ISBN 9781931971164.

[60] Long Zhang, Brice Morin, Philipp Haller, Benoit Baudry, and Martin Monperrus. A chaos engineering system for live analysis and falsification of exception-handling in the jvm. *IEEE Transactions on Software Engineering*, 47(11):2534–2548, 2021. doi: 10.1109/TSE.2019.2954871.

[61] Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 261–272, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350761. doi: 10.1145/3092703.3092731. URL https://doi.org/10.1145/3092703.3092731.

[62] 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. ISBN 9781450350853. doi: 10.1145/3132747.3132768. URL https://doi.org/10.1145/3132747.3132768.

[63] Xu Zhao, Kirk Rodrigues, Yu Luo, Michael Stumm, Ding Yuan, and Yuanyuan Zhou. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 565–581, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132778. URL https://doi.org/10.1145/3132747.3132778.