CSnake: Detecting Self-Sustaining Cascading Failure via Causal Stitching of Fault Propagations

Shangshu Qian
Purdue University
West Lafayette, Indiana, USA
shangshu@purdue.edu

Lin Tan
Purdue University
West Lafayette, Indiana, USA
lintan@purdue.edu

Yongle Zhang
Purdue University
West Lafayette, Indiana, USA
yonglezh@purdue.edu

Abstract

Recent studies have revealed that self-sustaining cascading failures in distributed systems frequently lead to widespread outages, which are challenging to contain and recover from. Existing failure detection techniques struggle to expose such failures prior to deployment, as they typically require a complex combination of specific conditions to be triggered. This challenge stems from the inherent nature of cascading failures, as they typically involve a sequence of fault propagations, each activated by distinct conditions.

This paper presents CSnake, a fault injection framework to expose self-sustaining cascading failures in distributed systems. CSnake uses the novel idea of *causal stitching*, which causally links multiple single-fault injections in different test workloads to simulate complex fault propagation chains. To identify propagation chains between faults, CSnake designs a *counterfactual* causality analysis of fault propagations – *fault causality analysis* (FCA): FCA compares the execution trace of a fault injection run with its corresponding profile run (i.e., running the same test without the injection) and identifies any additional faults triggered, which are considered to have a causal relationship with the injected fault.

To address the large search space of fault and workload combinations, CSnake employs a *three-phase allocation (3PA)* protocol of test budget that prioritizes faults with unique and diverse causal consequences, thereby increasing the likelihood of uncovering conditional fault propagations. Furthermore, to avoid incorrectly connecting fault propagations from workloads with incompatible conditions, CSnake performs a *local compatibility check* that *approximately* checks the compatibility of the path constraints associated with connected fault propagations with low overhead.

CSnake has detected 15 bugs that resulted in self-sustaining cascading failures in five widely deployed distributed systems, five of which have been confirmed with two fixed.

EUROSYS '26, Edinburgh, Scotland, UK

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

Keywords: Distributed System, Cascading Failure, Bug Detection, Fault Injection, Cloud Computing

ACM Reference Format:
Shangshu Qian, Lin Tan, and Yongle Zhang. 2026. CSnake: Detecting Self-Sustaining Cascading Failure via Causal Stitching of Fault Propagations . In *European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland, UK*. ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3767295.3769373

1 Introduction

Distributed systems are designed to tolerate component failures [28]. However, recent research has revealed self-sustaining cascading failures [41, 48, 64, 78], wherein a fault propagates across components and replicates itself through a self-reinforcing loop [48, 78]. These failures undermine the intended fault tolerance, resulting in widespread outages with severe consequences. For example, a self-sustaining cascading failure happened in Amazon AWS [1] on July 30th, 2024, significantly disrupting core AWS services, including AWS Lambda, EC2, and S3. The incident also brought down many dependent services, such as Whole Foods Supermarket, Amazon Alexa, and Goodreads [44].

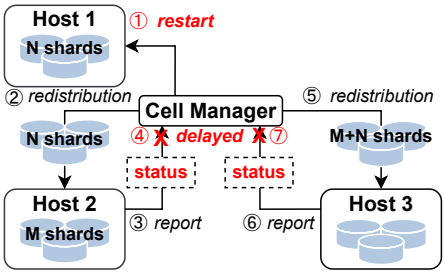

Figure 1. A real-world self-sustaining cascading failure from Amazon AWS. The Cell (Cluster) Manager manages a cluster of hosts (servers), each storing a set of data shards.

Such failures often require intricate combinations of conditions to manifest [78], as they typically involve a sequence of fault propagations, each activated by distinct conditions. For instance, as shown in Figure 1, the Amazon AWS incident [1] occurred in a cluster of servers (hosts) managed

by a cluster (cell) manager, with each server hosting a set of data shards. During a routine system upgrade, the cell manager (①) restarted some hosts (e.g., Host 1), and (②) redistributed their shards to other hosts (e.g., Host 2). Due to a latent bug in the load balancer, the cell manager mistakenly redistributed all *low-throughput shards* to a small number of hosts (Host 2). Consequently, these hosts started (③) sending abnormally large status reports that include metadata for all hosted shards. The increased size of these reports caused delays in both transmission and processing (④), causing the management system to misclassify these servers as unhealthy. The management system consequently removed these servers from the cluster, and (⑤) redistributed their shards to other servers, which causes the receiving servers (e.g., Host 3) to send large-sized reports (⑥) and subsequently be removed from the cluster (⑦). This failure involves two fault propagations, each requiring distinct conditions. First, for a server removal to propagate and trigger incorrect redistribution (① → ② and ④ → ⑤), the redistributed shards must have low throughput. Second, for incorrect redistribution to induce report delay and subsequent server removal (② → ④ and ⑤ → ⑦), the report size (or, the number of affected shards) must be sufficiently large.

Existing fault injection techniques [9, 19, 20, 29, 31, 38, 40, 56, 57, 63, 64, 68, 70, 71, 74, 75, 83, 86, 89] fall short of exposing such self-sustaining cascading failures before deployment, because they only inject faults into a limited set of manually selected (e.g., stress tests) or synthetically crafted workloads (e.g., benchmark workloads), which often **lack the required combination of conditions**. Conversely, if a workload satisfying both of the previously described conditions had been exercised, a traditional fault commonly used in existing fault injection frameworks – a server crash – would have exposed the AWS self-sustaining cascading failure before deployment. Missing triggering conditions is a known challenge to effective fault injection testing [31, 65]. Without prior knowledge of the targeted bug, it is exceptionally hard for the developers to manually create test workloads or rules for workload generators [43, 56, 60, 70] that meet all necessary conditions.

To tackle this challenge, we propose **Causal Stitching** to simulate complex fault propagation under complex conditions by **causally linking multiple single-fault injections in different workloads**. Each injection uncovers one step in the fault propagation chain. Our insight is that triggering one step in the chain requires less stringent conditions than triggering the whole chain, increasing the likelihood of achieving this through fault injection into existing workloads, such as the integration tests shipped with the system.

To illustrate, consider the two faults involved in the Amazon AWS self-sustaining cascading failure: node loss (f_1) and imbalanced shard redistribution (f_2). Their causal relationships can be identified through separate fault injection runs. Specifically, in a test case t_1 with low-throughput shards (condition c_1), injecting f_1 results in the triggering of an additional fault f_2, establishing the causal relationship $f_1 \rightarrow f_2$. Conversely, in a test case t_2 with a large number of shards hosted on individual nodes (condition c_2) , injecting f_2 (simulated via induced delay) leads to the manifestation of f_1, indicating a reverse causal relationship $f_2 \rightarrow f_1$. The self-sustaining cascading failure can be reconstructed by linking the causal relationship among the two faults as long as the workload and conditions in tests t_1 and t_2 are *compatible*, resulting in a causal cycle of $f_1 \rightarrow f_2 \rightarrow f_1$.

We could not apply this idea to the AWS incident due to the lack of access to the source code and test suite. However, our tool – CSnake– demonstrated its capability to detect a similar, previously unknown self-sustaining cascading failures in HBase [4], an open-source distributed database (§8.3). Notably, HBase's test suite does not contain a single workload that satisfies all the necessary triggering conditions; instead, CSnake identified and connected causal relationships across multiple test cases to reveal the failure.

To obtain the causal relationships between faults inside the system, we propose a *counterfactual* causality analysis of fault propagations – **fault causality analysis (FCA)**. FCA compares the execution trace of a fault injection run with its corresponding profile run (i.e., running the same test without the injection) and identifies any additional faults triggered, which are considered to have a causal relationship with the injected fault.

A major challenge faced by **Causal Stitching** is the vast number of combinations of faults and workloads. To efficiently explore the causal relationship between faults in such a massive search space, CSnake uses a **three-phase allocation (3PA) protocol** during test execution to maximize the number of causal relationships discovered under a fixed test budget. Intuitively, 3PA prioritizes injecting faults with unique and diverse causal consequences. This prioritization is guided by an *adaptive weighting algorithm*, which leverages runtime feedback from prior fault injection runs to estimate the potential of each fault to uncover new causal relationships, particularly conditional propagations. In addition, in the subsequent cycle detection phase, CSnake utilizes a **parallel beam search** to construct propagation chains, also applying this prioritization principle to favor faults with unique and diverse causal consequences.

Another challenge is the risk of linking causal relationships discovered in workloads with incompatible conditions. For example, suppose the causal relation $f_1 \rightarrow f_2$ is observed in test t_1 under a condition c_1, while $f_2 \rightarrow f_1$ in test t_2 happens under the negation of c_1, linking $f_1 \rightarrow f_2$ and $f_2 \rightarrow f_1$ into a single causal cycle is unsound, as the conditions required for each step are mutually exclusive. To reduce the risk while minimize the overhead, CSnake approximates such symbolic constraints with the fault's local program trace, including branch evaluation results and the call stack. CSnake performs a **local compatibility check** with such

approximated constraints before linking causal relationships, and assumes that if the traces leading to the same fault are similar in different tests (e.g., traces leading to f_2 in t_1 and t_2), the compatibility between tests exists.

In summary, this paper makes the following contributions:

- We propose **Causal Stitching**, a fault injection technique that detects complex fault propagations by causally linking multiple single-fault injections across workloads.
- To overcome the challenge of a massive search space, we propose a test budget allocation protocol (**3PA**) and a **parallel beam search** cycle detection algorithm, both of which prioritize faults with unique and diverse causal consequences.
- To reduce the risk of invalid linking of causal relationships, we implement a **local compatibility check** to eliminate incompatible conditions.
- We implement the fault injection and analysis framework, CSnake. We evaluate CSnake on the latest versions of five popular open-source distributed systems, i.e., HDFS 2.10.2 [10], HDFS 3.4.1 [11], HBase 2.6.0 [4], OZone 1.4.0 [5], and Flink 1.20.0 [3], where CSnake detects 15 new self-sustaining cascading failures, five of which have been confirmed with two fixed by developers. CSnake's source code is available at https://github.com/Purdue-PFL/CSnake.

2 Problem Statement

To introduce our problem definition, we first present a model of fault injection experiments of cascading failures extending the model of general fault injection experiments [75], as well as our fault model and causality model.

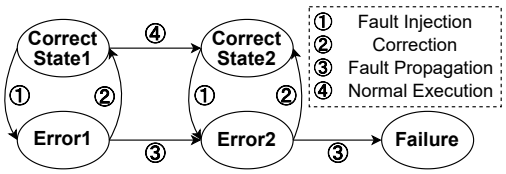

Figure 2. Model of fault injection experiments of cascading failures, characterized by different system states.

Model of Fault Injection. As shown in Figure 2, fault injection experiments are modeled using transitions between system states. During normal system execution (④), a *fault* can be injected (①) at different time points (Correct State1 and State2) to trigger an *error* (Error1 and Error2). The injection can be performed in multiple experiments with different *workloads* (i.e., system inputs). An error could be masked or corrected (②) by fault tolerance mechanisms such as replication [12] and recomputation [88]. Under special conditions, it could propagate (③) and corrupt other parts of the system state to *cause* additional errors. Such propagations (e.g., Error1 → Error2 and Error2 → Failure) may require distinct *conditions* to be *activated*. A *condition* is a logical predicate over the system state.

Compared to the general fault injection model [75], our model captures the causal relationships between faults across **multiple** fault injection experiments. Specifically, in one experiment, a fault (f_1) injected into Correct State1 triggers Error1 and subsequently *causes* Error2 (without leading to Failure). In a separate experiment, a fault (f_2) injected into Correct State2 triggers Error2 and subsequently *causes* Failure. The model captures the possible fault propagation Error1 → Error2 → Failure, provided that the activation conditions for each causal relationship are *compatible*. The definitions of *fault*, *causality*, and *compatibility* are provided below.

Fault Model. Traditionally, *error* refers to an incorrect system state, while *fault* refers to its cause, such as software bugs and hardware faults [75]. In the remainder of this paper, we use *fault* and *error* interchangeably, because we perform a specific type of fault injection – *software-implemented fault injection* [75], which injects the *effects* of a fault, such as exceptions and delays, to simulate *errors* directly and accelerate the experiment, instead of injecting the actual faults.

Causality Model. To capture the causal relationship between *faults* (*errors*), we use *counterfactual causality*: f_1 is a counterfactual cause of f_2 if and only if f_2 would not occur unless f_1 occurs. According to the ladder of causation [76], *counterfactual causality* offers the strongest bond between cause and effect. To the best of our knowledge, although recently utilized in root cause localization [34, 55], *counterfactual causality* has not been used in fault injection to analyze how faults propagate. Existing analysis of failure propagation [62, 64] utilizes the happens-before relationship [64] and program dependencies [62], both of which are weaker forms of causality.

Problem Definition. Given a set of workloads $W = \{w_1, w_2, ...\}$, our main goal is to identify the causal relationships between faults ($f_1 \rightarrow f_2$) across workloads, and connect *compatible* identified causal relationships to detect *cycles*, where a fault causes itself through a chain of connected causal relationships.

Because the causal relationships may be identified in different workloads (e.g., $f_1 \rightarrow f_2$ identified in w_1 and $f_2 \rightarrow f_3$ in w_2), blindly connecting them could result in invalid causal chains when the workloads are *incompatible* – the *conditions* required to activate $f_1 \rightarrow f_2$ in w_1 and $f_2 \rightarrow f_3$ in w_2 are mutually exclusive. Therefore, we require the connected causal relationships to be *compatible*, meaning the conjunction of their activating conditions is satisfiable.

For simplicity of the paper, in a causal relationship $f_1 \rightarrow f_2$ identified from w_1, we name f_2 the *interference* of f_1 on the system due to the fault injection. The causal relationship also forms a *fault propagation chain* of length 1 from f_1 to f_2.

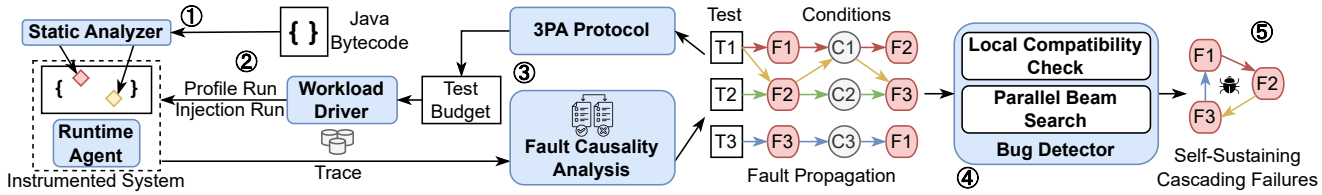

Figure 3. Overview of CSnake. Blue boxes are components of CSnake.

3 CSnake Overview

CSnake operates by identifying causal relationships between faults in distributed systems and linking them to form potential cycles. It simulates chains of fault propagations in complex workloads quasi-statically by combining the causal relationships derived from fault injections on simpler workloads.

At the core of CSnake is a counterfactual causality analysis. Counterfactual causality analysis involves conducting two experiments: one with a fault injected to observe its impact on the system execution (referred to as the *interference*), and the other without the fault injected to observe the system's execution (referred to as the counterfactual – i.e., what would the system execution be had the fault not been injected). Specifically, we compare the execution trace of a fault injection run with its corresponding profile run (i.e., running the same test without the injection) and identify any additional faults triggered, which are considered to have a counterfactual causal relationship with the injected fault.

Workflow. Figure 3 illustrates the workflow of CSnake. (①) Given the Java bytecode of the target system, CSnake's *static analyzer* identifies system-specific fault injection points, and instruments fault injection and monitoring hooks. At runtime, CSnake's *runtime agent* performs fault injection and monitoring when a hook is encountered.

During fault injection experiments (②), a *workload driver* selects combinations of faults and workloads (integration tests shipped with the target system) to execute both the fault injection run and the corresponding fault-free profile run (counterfactual experiment). Note that, for each fault, CSnake only uses the set of tests that can reach the fault during execution in fault injection runs.

Next, the *fault causality analysis* module (③) compares the execution traces from the profile and injection runs to identify counterfactual causal relationships – i.e., *fault propagations*. These causal relationships are used by the *three-phase allocation (3PA) protocol* to guide future workload and fault selection via feedback to the workload driver.

The discovered causal relationships are also forwarded to CSnake's *bug detector* (④), which connects *compatible* relationships and performs a *beam search* for cycle detection to identify self-sustaining cascading failures (⑤).

In the following sections, we first explain (§4) how to instrument faults and perform the core counterfactual causality analysis, and then (§5) how to efficiently allocate testing resources, and finally (§6) how to connect compatible causal relationships and detect cycles.

4 Fault Causality Analysis

This section details the types of faults to inject (§4.1), the fault injection process (§4.2), and the approach for establishing causal relationships between faults (§4.3).

4.1 What faults to inject?

A recent study [78] on self-sustaining cascading failures from open-source systems reveals that self-sustaining cascading failures are often caused by functional and performance interferences between requests and error handlers. Functional interferences typically manifest as exceptions and errors captured by system-specific error detectors, and performance interferences typically manifest as contention. Following this observation, CSnake injects three types of faults.

Exception. A target system can encounter three types of exceptions during its execution: (1) system-specific exceptions – exceptions thrown explicitly inside the target system's source code, (2) library function exceptions – exceptions thrown by a library or native function, (3) unchecked (runtime) exceptions – exceptions that are not required to be explicitly handled (by try/catch blocks in JVM-based systems) or declared explicitly in function signatures.

CSnake injects system-specific exceptions and library function exceptions, because they occur at limited, explicitly-declared program locations, while unchecked exceptions can happen at arbitrary locations. Note that CSnake still injects an unchecked exception if it is explicitly thrown in the system's code – e.g., an `IllegalArgumentException` is often thrown when a request with invalid input is received. In practice, CSnake filters out reflection-related and security-related exceptions, as they tend to terminate or hang the system rather than propagate. CSnake also ignores exceptions only reachable from tests.

Contention. Though contention can happen at arbitrary program locations, CSnake only considers loops for contention injection to simulate resource-intensive tasks causing contention, due to their association with performance issues discovered by existing studies [54, 64, 78]. CSnake uses iteration count of the workload-related loops to capture

```
1  for (int i = 0; i < 10; i++) { // Loop 1
2    sock.read(); // TP1
3    if (condition1) {// TP2; MP1
4      throw new IOException();
5    }
6    for (int j = 0; j < 10; j++) { // Loop 2
7      if (foo()) { // TP3; MP2
8        throw new IOException();
9      }
10   }
11 }
12
13 boolean foo() { // NP1
14   fis = new FileInputStream(path); // TP 4
15   if (fis.read() == -1) { // MP 3; TP 5
16     return false;
17   }
18   return true;
19 }
```

Figure 4. Pseudo-code demonstrating the injection and monitor points. "TP" means throw point and "NP" negation point. "MP" means monitor point (used and explained in §6.2).

contention induced by increased workload (§4.3), similar to the approach adopted by Li et al. [64].

CSnake adopts a loop scalability analysis to filter out loops unlikely to cause performance issues. CSnake excludes loops with a constant upper bound on their iteration count, detected by a best-effort data flow analysis to track the loop guard condition to a constant. In addition, CSnake ranks the size of code reachable recursively starting from a loop and identifies loops involving I/O operations, to exclude loops that have a short execution (i.e., lowest ranked 10% of loops) and do not perform I/O.

System-Specific Error. Many distributed systems implement system-specific error detectors, such as status checks and health monitoring, without using exceptions. Such system-specific error detectors are often implemented as functions with a boolean return value. For example, in HDFS, a NameNode thread checks if a DataNode's heartbeat is received within a timeout limit using a function `node.isStale()`.

CSnake filters out boolean-returning functions if they are unlikely to be a system-specific error detector. For example, CSnake filters out such boolean-returning functions in JDK libraries, such as `contains()` function of Java collection type objects (detailed filtering criteria in §7). CSnake's fault filtering criteria is designed to be conservative to avoid meaningful fault being skipped accidentally. The injection points that do not impact system execution will be automatically deprioritized during fault injection experiments by the test budget allocation protocol (§5).

4.2 How to inject faults?

CSnake's static analyzer identifies locations to inject faults and instruments fault injection hooks statically. The hook transfers the control to CSnake's runtime agent. During fault injection experiments, CSnake dynamically injects faults when corresponding hooks are encountered. We use a Java-style pseudo-code with eight fault locations (shown in Figure 4) to explain how fault injection is performed.

Exception Injection. For system-specific exceptions, as they are typically thrown inside an if-statement (lines 4 and 8), CSnake injects a one-time throw of the same exception when the if-statement (lines 3 and 7, referred to as *Throw Point*) is reached. For library function exceptions, CSnake injects a one-time throw of the exception declared in the function signature at the invocation site of these functions (lines 2, 14, and 15). Exceptions are constructed at runtime using their simplest constructor.

Contention Injection. For potentially non-scalable loops identified by CSnake (e.g., line 1 and 6), CSnake injects a predefined amount of **spinning delay** into each loop iteration before the first line of the loop body executes to simulate potential contention induced by executing a large number of iterations of this loop. Due to the way we inject contention, we refer to such injection as *delay injection*.

Each delay injection is performed seven times with varying length (100ms to 8s) to maximize the discovery of causal relationships between faults. The system is configured with *reduced timeouts* (10-20 seconds) to make it more sensitive to additional workload.

The delay and timeout values are determined empirically to maximize the impact of delay injections. We first lower system timeout configurations to a safe range of 10-20s via trial and error, which ensures normal system functionality being preserved (i.e., all unit and integration tests still pass). Based on this threshold, we select seven static delay values between 100ms and 8s that are likely to cause timeouts when injected repeatedly inside loops. We identify relevant configuration items using the keywords of "timeout" and "interval". The timeout adjustment process typically takes a student 30 minutes per system, and are not essential for CSnake's effectiveness, as delay-related causal relationships can still be observed with default settings.

System-Specific Error Injection. Because system-specific errors are captured by system-specific error detectors, whose return values are boolean typed, CSnake negates the return value of these functions (which is referred to as *Negation Point*) before the execution returns to its caller to simulate the effect of such faults (`foo()` at line 13). We refer to the injection of system-specific error as a *negation injection*.

4.3 How to establish causal relationships?

CSnake records encountered faults during the injection run and profile run, and compares them to identify additional faults triggered to establish causal relationships. We categorize the encountered faults into the following categories:

1. **Execution Trace Interference**. For **exceptions**, CSnake monitors whether the throw statement is reached.

```
while (shouldRun()) { // Loop 1
    resp = sendHeartBeat();
    // Loop 2
    for (DataNodeCommand cmd: resp.getCommands()) {
        processCommand(cmd);
    }
    // ...
    List<BlockReport> reports = new ArrayList<>();
    // Loop 3
    for (Map.Entry kv: perVolumeBlocks.entrySet()) {
        reports.add(convertFormat(kv));
    }
    // ...
    for (BlockReport report: reports) {
        nnRpcStub.blockReport(report);
        // ...
    }
}
```

Figure 5. Pseudo-code demonstrating delay in nested loops, code simplified from `BPServiceActor` in HDFS.

For **system-specific errors**, CSnake monitors whether the return value of the error-detector function is negated.

2. **Iteration Count Interference**. For **contention**, CSnake monitors whether any loop's iteration count *statistically* increases compared to the profile run, as an indicator for contention. The loop iteration count serves as a good indicator for the amount of workload processed [64]. We use one-sided t-test [81] with a p-value of 0.1 for statistical significance.

CSnake executes each profile run and injection run five times to reduce the impact of **non-determinism** in the system execution. This also allows us to use statistical tests on loop iteration counts.

Injecting f_1 may trigger multiple additional faults, stemming from both interference types. After all the injection runs for f_1, CSnake collects a list of additional faults triggered $[f_2, f_3, ..., f_n]$.

Nested and Consecutive Loops. One special case of iteration count interference is the handling of nested and consecutive loops in the system. Three types of compositions of workload related loops exist: 1) independent loops, 2) nested loops, and 3) consecutive loops. We observe that 2) nested and 3) consecutive workload-related loops are often used in batch processing. In such loops, a delayed sub-request can propagate delays to its parent request (the batch) and the next sub-request in the same batch. For example, in a batched RPC request, a delayed sub-request can time out the entire call as well as the next sub-request.

CSnake identifies the parent-child loop pairs to establish this causal relationship and expand the impact scope of contention injection. Figure 5 shows the pseudo-code demonstrating contentions in the nested loops. If injection f_1 increases iterations in loop 2 (L_2), its parent (L_1) and sibling (L_3) can also be affected. We use $L_2 \xrightarrow{\texttt{ICFG}} L_1$ to represent the former ("I" for "Inverse") and $L_1 \xrightarrow{\texttt{CFG}} L_3$ to represent the latter.

Table 1. Summary of the causal relationships between faults.

Type	Injected Fault	Additional Fault
$\xrightarrow{\texttt{E(D)}}$	Delay	Exception; Negation
$\xrightarrow{\texttt{S}^+\texttt{(D)}}$	Delay	Delay
$\xrightarrow{\texttt{E(I)}}$	Exception; Negation	Exception; Negation
$\xrightarrow{\texttt{S}^+\texttt{(I)}}$	Exception; Negation	Delay
$\xrightarrow{\texttt{ICFG}}$	$f_1 \xrightarrow{\texttt{S}^+\texttt{(D/I)}} f_2 \xrightarrow{\texttt{ICFG}} f_2'$ only	
$\xrightarrow{\texttt{CFG}}$	$f_1 \xrightarrow{\texttt{S}^+\texttt{(D/I)}} f_2 \xrightarrow{\texttt{ICFG}} f_2' \xrightarrow{\texttt{CFG}} f_2''$ only	

Summary for Fault Causality. Table 1 summarizes all the six causal relationships between different types of faults. The first four are differentiated by the injected fault and the additional fault triggered. The last two are special causal relationships handling nested loops, extending the impact range of contentions.

1. $f_1 \xrightarrow{\texttt{E(D)}} f_2$: Injecting a delay f_1 into a loop causes an additional exception or negation f_2. Delay f_1 has an execution trace interference f_2 on the system (hence the name `E(D)`).
2. $f_1 \xrightarrow{\texttt{S}^+\texttt{(D)}} f_2$: Injecting a delay f_1 into a loop causes an additional delay (f_2) in another loop. Delay f_1 has an iteration count interference f_2 on the system (name $\texttt{S}^+$ indicates a statistically significant increase).
3. $f_1 \xrightarrow{\texttt{E(I)}} f_2$: Injecting an exception or negation f_1 causes an additional exception or negation f_2.
4. $f_1 \xrightarrow{\texttt{S}^+\texttt{(I)}} f_2$: Injecting an exception or negation f_1 causes an additional delay (f_2) in another loop.
5. $f_2 \xrightarrow{\texttt{ICFG}} f_2'$: f_2 is an additional delay caused by the injection f_1, affecting its parent loop (f_2').
6. $f_2' \xrightarrow{\texttt{CFG}} f_2''$: Following $f_2 \xrightarrow{\texttt{ICFG}} f_2'$, the parent loop delay (f_2') further affects the sibling loop (f_2'').

5 Test Budget Allocation

A challenge faced by the approach of **causal stitching** is the vast number of combinations of faults and workloads. For example, HDFS has about 3,000 source code locations where exceptions can be thrown and 2,000 loops where delays can be injected and thousands of integration tests. Combining injected faults with different workloads can result in tens of millions of possible fault injection scenarios. In this section, we explain CSnake's **three-phase allocation (3PA) protocol** of test budget, which maximizes the discovery of causal relationships within a fixed test budget. [1]

5.1 Principles of Test Allocation

The 3PA protocol leverages two principles to prioritize faults based on the uniqueness and diversity of their causal impact: 1) injecting *causally equivalent* faults to diverse workloads, and 2) extending *conditional* causal relationships.

[1] A formal definition of the 3PA protocol is in §A.

Causally Equivalent Fault. Since different faults (e.g., f_1 and f_2) may trigger similar interferences (i.e., additional faults), we consider such faults (f_1 and f_2) *causally equivalent* with respect to their consequences. For example, in a try-catch block, multiple exceptions in the try block might be handled by the same catch block, in which triggers an additional fault. In such cases, it is redundant to inject *causally equivalent* faults into the same workload. Instead, CSnake prioritizes to inject them in different workloads in order to maximize the discovery of diverse fault propagations.

Conditional Causal Relationship. Because self-sustaining cascading failures are often formed by fault propagations activated by specific conditions, CSnake allocates test budget to increase the probability to extend *conditional* causal relationships between faults. CSnake detects such *conditional* causal relationships by checking if a fault causes different faults in different workloads. For example, if f_1 always causes f_2 in different workloads, $f_1 \rightarrow f_2$ is considered *unconditional*, while if f_1 causes f_2 in one workload but f_3 in another, then $f_1 \rightarrow f_2$ and $f_1 \rightarrow f_3$ are considered *conditional*. CSnake prioritizes to allocate test budgets to trigger the resulting faults (f_2 and f_3) of such conditional causal relationships.

5.2 Three-Phase Allocation Protocol

Following the above principles, CSnake allocates the test budget in three phases. In phase one (**causally equivalent fault detection**), CSnake conducts a preliminary exploration by running each fault injection against one test workload, and clusters faults causing similar interferences in order to detect *causally equivalent* faults. In phase two (**causality exploration**), CSnake picks a fault from each set of *causally equivalent* faults and injects it into diverse workloads to explore its causal relationships with other faults. In phase three (**conditional-causality-guided causality extension**), CSnake uses the result from phase two to detect *conditional* causal relationships, and prioritizes to inject faults that can extend such *conditional* causal relationships. The total test budget is calculated using the number of fault locations in each system, currently as $4 \times |\mathbb{F}|$, where $\mathbb{F}$ is the set of faults. Phase one makes up the first 25% of the budget. Phase two gets 50% of the budget, while phase three uses the remaining 25%.

Phase One: Causally Equivalent Fault Detection. In this phase, for each fault f_i, CSnake picks the workload t_{i_1} that reaches f_i's program location and has the highest code coverage. The rationale is that such a test is more likely to reveal the most diverse interferences of f_i. Different faults may be injected into different workloads since a single test may not cover all fault locations. CSnake then performs the fault causality analysis to obtain all additional faults triggered. We use $I(f_i, t_j)$ to represent a list of additional faults triggered by injecting f_i to t_j.

Injections that do not affect system execution will be deprioritized automatically. These non-impactful injections will be clustered together, and assigned the lowest allocation weight in the remaining phases. This complements our conservative static filtering of the injection points (§4.1).

CSnake uses an IDF-based (inverse document frequency [69], more details in §A.1) clustering algorithm to detect *causally equivalent* faults. IDF is commonly used in text mining tasks for applying weights to the words [25]. It excels at reducing noises induced by common words such as "a" and "the". CSnake uses IDF to assign a weight to each fault f in the fault corpus $\mathbb{F}$. The intuition is that, similar to common words in text mining, if a fault is frequently triggered by various other faults (e.g., inside a utility function), it is less useful for determining the similarity of interferences $I(f_i, t_j)$ from different injections. CSnake uses such weights to determine the similarity of the interferences caused by different injections.

Specifically, CSnake vectorizes each $I(f_i, t_{i_1})$ as $V(f_i, t_{i_1}) = \text{IDF}_\text{v}(I(f_i, t_{i_1}), \mathbb{F})$, resulting in a real vector of length n, with each element ranging from 0 to 1 (i.e., $V \in [0, 1]^n$). CSnake then performs a hierarchical clustering [58] of the faults in $\mathbb{F}$ using the cosine distance [69] between all vectorized interferences $V(f_i, t_{i_1})$. We select hierarchical clustering due to their explainable nature, as required in biology [61] and medical [91] researches. We use cosine distance instead of euclidean distance due to their insensitive to vector length (i.e., the number of additional faults triggered in this case). The cosine distance here ranges from 0 to 1 because all IDF vectors are positive.

By the end of phase one, each fault f_i is clustered in to a group G_j with other faults having similar interferences on the system once injected. This phase makes up 25% of the test budget, providing an initial understanding of each fault's interference.

Phase Two: Causality Exploration. In phase two, CSnake distributes test budgets among fault clusters G_i in a round-robin manner to perform injection and explore their inteferences. This approach allocates equal test budgets to fault clusters, not individual faults, avoiding redundant injections that yield similar interferences.

In both phase two and three, each time a cluster receives a test quota, CSnake **randomly** selects one fault from that cluster and injects it into a new workload. This is to compensate for potential inaccuracies in phase one due to limited test budget. For example, injecting f_1 and f_2 both causes f_3 in phase one, but f_1 can cause f_4 in another workload which is unfortunately not used in phase one. Using random selection, every fault within a cluster have a chance be injected into a new workload.

After completing all the injection runs, CSnake performs fault causality analysis again to identify additional faults $I(f_i, t_j)$. A second IDF vectorizer is trained with the data

from both phases to convert $I(f_i, t_j)$ to $V(f_i, t_j)$, which is used in prioritization in the next phase.

CSnake prioritizes to inject faults that can cause conditional interferences to more workloads, in order to extend conditional causal relationships. It measures the *diversity* of the causal consequences (interferences) of faults in a cluster using an intra-cluster interference similarity score (SimScore(G_i)). CSnake prioritizes to inject faults from clusters with a lower score, which increases the chance for the injection to trigger conditional interferences. The score is the average pairwise cosine distance between all vectorized interference results $V(f_i, t_j)$ obtained from the injection experiments conducted in cluster G_i during phase one and two. SimScore(G_i) ranges from 0 to 1. A value of 1 indicates that all faults in cluster G_i trigger the same set of additional faults among all injection runs. The score is then used in phase three.

Phase Three: Conditional-Causality-Guided Causality Extension. Using the SimScore(G_i) calculated in phase two, CSnake performs a weighted random allocation favoring clusters with more conditional causal consequences. Lower similarity indicates more conditional causal consequences and thus a higher allocation weight. The allocation weight for cluster G_i is defined as max (ϵ, 1 − SimScore(G_i)). Each cluster has a minimum weight ϵ of 0.01, ensuring every cluster receives some budget, even with perfectly matched intra-group interference results.

Test Budget Transfer. In phase two and three, test quotas are *transferable* between clusters. In phase two, if a cluster exhausts its (fault, test) combinations before using up its quota, the remaining quota is randomly transferred to a larger cluster for more thorough exploration. In phase three, any unused budget will be transferred to clusters with a smaller allocation weight.

Text Budget Selection. We recommend running a minimum of $4 \times |\mathbb{F}|$ tests under the 3PA protocol. In phase one, each fault needs one test to explore the potential interferences of the injection on the system. In phase two, each fault is on average injected into two additional test cases, which enables the intra-group similarity calculation. We set a number of 2 instead of 1 in this phase just in case one fault is injected into a similar test workload as phase one. In phase three, we naturally allocate each fault one additional test.

6 Detect Self-Sustaining Cascading Failures

This sections details how CSnake stitches causal relationships to form fault propagation chains (§6.1), how to avoid incompatible stitching (§6.2), and how CSnake searches for cycles to detect self-sustaining cascading failures (§6.3).

6.1 Stitching Causal Relationships

CSnake stitches causal relationships when the resulting fault in one causal relationship (f_2 in $f_1 \rightarrow f_2$) matches the starting fault in another (f_2 in $f_2 \rightarrow f_3$). In particular, based on the type of the fault (f_2) used in stitching, CSnake can form four types of connections:

1. $f_1 \xrightarrow{\text{E(D/I)}} f_2 \xrightarrow{\text{E/S}^+\text{(I)}} f_3$: The matching fault f_2 is an exception or negation. Injecting f_1 has an execution trace interference f_2 on the system. f_1 and f_3 can be of any type.
2. $f_1 \xrightarrow{\text{S}^+\text{(D/I)}} f_2 \xrightarrow{\text{E/S}^+\text{(D)}} f_3$: The matching fault f_2 is a delay on a loop. Injecting f_1 has an iteration count interference f_2 on the system. f_1 and f_3 can be of any type. For delays in nested loops performing batch processing, there are two variants:
   a. $f_1 \xrightarrow{\text{S}^+\text{(D/I)}} f_2 \xrightarrow{\text{ICFG}} f_2' \xrightarrow{\text{E/S}^+\text{(D)}} f_3$: The delay f_2 propagates to its parent loop (f_2'), which is the injected fault of the next injection and causes f_3.
   b. $f_1 \xrightarrow{\text{S}^+\text{(D/I)}} f_2 \xrightarrow{\text{ICFG}} f_2' \xrightarrow{\text{CFG}} f_2'' \xrightarrow{\text{E/S}^+\text{(D)}} f_3$: The delay f_2 propagates to its sibling loop (f_2''), which is the injected fault of the next injection and causes f_3.

```
1  public ReplicaHandler createTmp(...) {
2    do {
3      if (current == lastFoundReplica) { ... };
4      // ...
5      // Fault F2 (Injected)
6      if ((current.genStamp() >= b.genStamp()) ||
       ↪  !isTransfer)
7        // Fault F2 (Triggered by F1 injection)
8        throw new ReplicaAlreadyExistsException(...);
9    } while (true);
10 }
11
12 class BlockReceiver {
13   BlockReceiver(...){
14     data.createTmp(...);
15   }
16 }
```

Figure 6. Pseudo-code demonstrating state compatibility. The fault is located inside a loop of `createTmp()`, which is invoked from `BlockReceiver()`. Code simplified from `FsDatasetImpl` in HDFS.

6.2 Local Compatibility Check

When connecting causal relationships of faults obtained from different tests, CSnake performs a compatibility check between the tests involved. Theoretically, to check for incompatible propagation activation conditions across tests (and skip stitching), a symbolic constraint collection (i.e., path condition collection used in symbolic execution techniques [27, 32]) should be applied during the fault propagation: collect the path conditions c_1 and c_2 during the propagation of $f_1 \rightarrow f_2$ in test t_1 and $f_2 \rightarrow f_3$ in test t_2 respectively, and check the satisfiability of their conjunction $c_1 \wedge c_2$.

To minimize the overhead, CSnake approximates this analysis by checking whether 1) the **local** execution trace associated with the fault used in stitching (f_2) and 2) the call stack match with their counterparts in the other test. In

practice, we find CSnake's strategies successfully eliminates most incompatible workloads.

To facilitate the compatibility check, in injection runs, CSnake records call stack and execution trace as follows:

1. *Execution trace recording*. CSnake records the branch statements and the evaluation results of the conditions **locally** in the fault's enclosing loop or function as the path conditions. For example, in Figure 4, the three `ifs` (line 3, 7, and 15) serve as monitor points for such branch statements.

2. *Call stack recording*. An error occurred in the same function but at different call sites may represent errors for different types of requests, which is invalid when stitched together. To address this, CSnake records the closest two levels of call stack for each loop iteration, exception, and boolean return value (excluding the enclosing function itself). This method resembles the 2-call-site sensitivity in pointer analysis, a common method balancing speed and accuracy [49, 66, 84]. Note that the scope of execution trace and call stack used in the comparison can both be adjusted.

Figure 6 shows an example for the compatibility check. The fault f_2 (exception on line 8) used in stitching is inside a loop of `createTmp()`. CSnake checks the following two things before stitching $f_1 \rightarrow f_2$ and $f_2 \rightarrow f_3$ together: 1) (call stack) `createTmp()` must be both invoked from `BlockReceiver()`; 2) (execution trace) the traces of line 3-5 in the fault-happening iteration of the fault-enclosing loop match between tests. If the fault is not within a loop, we use the trace of the enclosing function (e.g., `BlockReceiver()`) instead. Because delay injection is injected at the beginning of all iterations of a loop, CSnake conservatively checks for matching traces in any loop iterations between tests.

6.3 Searching for Self-Sustaining Cascading Failures

Due to the large search space and the need to incorporate the local compatibility check, CSnake uses a customized **parallel beam search** to find cycles, as shown in Algorithm 1. The algorithm starts from all causal relationships obtained from the three-phased fault injection, forming propagation chains of length 1 (line 2). Each while-loop iteration (line 4) represents one-level on the search tree, where one edge `e` is connected to each propagation chain `c` in the queue (lines 6-13). Before connection, CSnake performs the compatibility check (line 17) between the last edge in the chain and the new edge (line 8). If each chain cycles back to its beginning (line 10), CSnake reports a potential self-sustaining cascading failure.

At each level of the beam search, only B active chains are kept, sorted by the average intra-cluster interference similarity score (§5.2) of the injected faults in the chain (lines 14-15). The intuition is that CSnake favors self-sustaining cascading failures involving complex error handling logic, which developers might overlook during testing. Suppose each chain C_i has r fault injections $f_{k_1}, f_{k_2}, ..., f_{k_r}$, and each fault f_{k_j} belongs to fault cluster G_j. The score used in ranking is defined as: $\text{Score}(C_i) = \Sigma_{j=0}^{r} \text{SimScore}(G_j)/\|C_i\|$. Chains with lower intra-cluster similarity are kept, as they likely involve conditional causal relationships.

Algorithm 1: Parallel Beam Search

```
Data: All causal relationships between faults 𝔼
Data: Beam size B
Result: All self-sustaining cascading failures ℂ
1  Function beamSearch(𝔼, B)
2      queue ← e ∈ 𝔼;
3      ℂ ← ∅;
4      while queue ≠ ∅ do
5          next ← ∅;
6          for c ∈ queue do in parallel
7              for e ∈ 𝔼 do in parallel
8                  if match(c.lastEdge, e) then
9                      new ← c.append(e);
10                     if isCycle(new) then
11                         ℂ.add(new)
12                     else
13                         next.add(new)
14         sort(next);
15         queue ← next.subList(0, B);
16     return ℂ;
17 Function match(edge1, edge2)
18     return edge1.interference == edge2.injectedFault &
         isCompatible(edge1.state, edge2.state);
19 Function isCycle(c)
20     return match(c.lastEdge, c.firstEdge);
```

CSnake perform the beam search on the entire fault causal space explored by the fault injection. Although there is no limit on the number of faults in each chain, chains cannot grow indefinitely as the fault injection runs in the chain must remain compatible.

Clustering Reported Cycles. Because in 3PA's phase two and three, CSnake randomly picks a fault in a cluster of *causally equivalent* faults to perform injection, our detection algorithm can identify equivalent self-sustaining cascading failures which contains *causally equivalent* faults. For example, suppose CSnake reports two cycles "$f_1 \rightarrow f_2 \rightarrow f_1$" and "$f_3 \rightarrow f_2 \rightarrow f_3$", while f_1 and f_3 are from the same fault cluster G, the two cycles reported are likely the same bug due to f_1 and f_3's similar interference on the system. CSnake automatically clusters cycles found in the beam search based on the fault clusters (§5.2) involved in the cycle.

7 Implementation Details

CSnake is implemented with over 16,000 lines of Java code, 4,000 lines of Python, and 700 lines of shell scripts. The static analyzer uses WALA [18]. The instrumentor and runtime agent are based on a customized version of Byteman [17] for dynamic instrumentation. The test runner uses a modified version of JUnit [14], which works with CSnake's instrumentor and the runtime agent to dynamically inject faults

Table 2. Number of injection points, monitor points, and integration tests in each system.

System	Loop	Exception	Negation	Branch	Test
HDFS 2	2067	2316	770	26941	2674
HDFS 3	2736	2707	974	36745	4426
HBase	3227	2369	1002	38192	4436
Flink	2860	2619	447	27947	2036
OZone	1361	1395	395	18212	1219

before each test. Checkpoint/Restore In Userspace (CRIU) [8] snapshots the JVM, speeding up the test initialization.

System-Specific Error Filtering. Except for filtering out system-agnostic functions with boolean return value such as Java collection operations (§4.1), CSnake filters out more system-specific errors according to the following criteria:

1. The boolean return value of the function only involves variables declared as `final`. Such variables are often related to system configurations. Configuration errors are not within the scope of this study. Instead of hunting for configuration errors, CSnake searches for fault propagation under valid configurations in different workloads.

2. The return value is constant or never used in the program. A negation placed on those functions will not have an impact on the system.

3. The return value is calculated only from primitive-type variables. This happens when the function is a utility function used in implementations of classic algorithms (e.g., `isSorted()` in a sorting algorithm). A negation will cause incorrect calculations, which is not a fault under CSnake's scope.

Additional implementation details, including a dynamic call graph construction and a customized JUnit framework, can be found in §B.

8 Evaluation

We evaluate CSnake on the latest versions of five popular distributed systems, namely, HDFS 3.4.1 [11], HDFS 2.10.2 [10], HBase 2.6.0 [4], Flink 1.20.0 [3], OZone 1.4.0 [5]. The fault injection tests and beam search are performed on two Ubuntu 22.04 servers: one with two Intel Xeon Gold 5220R CPUs and 512 GB of memory, and the other with two AMD EPYC 7313 CPUs and 256 GB of memory. Each profile and injection run is executed in a Docker container with Java 1.8. Each JVM instance is limited to 6 CPU cores and a heap size of 32GB.

Table 2 summarizes the number of injection points, monitor points, and test workloads identified in each system. In all the evaluations, we use a beam size of 5 million chains.

8.1 New Self-Sustaining Cascading Failures

Table 3 shows the 15 new self-sustaining cascading failures detected by CSnake in all five distributed systems, many of which involve retries and the failure recovery logic. This aligns with the findings of Qian et al. [78] on self-sustaining cascading failures. Five of them have been confirmed, with two fixed. Two bugs detected in HDFS 2 are also detected in HDFS 3, we do not list them in Table 3 due to duplication.

Table 3 also details the characteristics of the fault injection runs for each bug. Column "Cycle" shows the number of faults injected of each type. Column "Alloc." shows the test budget allocation phase after which all the causal relationships of each bug are discovered. All three phases of the 3PA protocol contributes to detecting new bugs.

To further demonstrate 3PA protocol's effectiveness, we compare it with a random allocation protocol, running the same number of fault injection runs as the 3PA protocol with randomly selected (fault, test) combinations. A "✓" in column "Rnd.?" of Table 3 means that the bug can be detected using random allocation. The random selection allocation protocol, albeit effective in some systems (i.e., Flink), generally underperforms the 3PA protocol in CSnake.

Most new failures detected by CSnake require only one delay injection, with varying numbers of exception and negation injections. This is in line with the bug dataset provided by Qian et al. [78], where majority of the cases only requires a single contention to be triggered.

Self-sustaining cascading failures reported by CSnake that involve multiple delay injections generally contribute to the false positive cases. However, most of the false positive cases are valid fault propagations, but categorized as false positive because they are known (and accepted by the developer) contentions between operations in the system (e.g., contention between the HDFS clients performing read and write). Details will be discussed in the §8.4.

8.2 Comparison with Alternative Strategy

We compare CSnake with a naive strategy that injects a single fault into a system and monitors whether it causes itself (e.g., delays a single loop in the system and monitors if its iterations increase). This is to evaluate whether the triggering conditions of each detected bug span multiple tests.

Column "Alt.?" in Table 3 shows that about 73% (11 out of 15) of the self-sustaining cascading failures detected by CSnake cannot be triggered by the naive strategy. This underscores the effectiveness of CSnake's causal stitching, the complexity needed to trigger these failures, and the limitations of single-fault injection tools. Even when the naive strategy works, CSnake offers detailed insights into the cycle's behavior.

8.2.1 Comparison with Existing Fuzzing Techniques.

In addition to the naive strategy above, we compare CSnake with blackbox fuzzing techniques. Specifically, we compare with Jepsen [9] and its Python reimplementation, Blockade [7]. Jepsen is a Clojure-based blackbox distributed system

Table 3. All 15 new self-sustaining cascading failures detected by CSnake. In general, "DN" stands for DataNode, "NN" stands for NameNode, and "IBR" stands for incremental block report. "Cycle" column shows the types of faults in each cycle. "Alloc." column shows the phase number in 3PA protocol that each bug is detected in. A "✓" in column "Rnd.?" means that the bug is detectable under random allocation of test budgets. "Alt.?" column shows whether the bug can be triggered by the strategy in §8.2. "JIRA#" column lists the bug ID in Apache issue tracker.

System	#	Delayed Task	Other Faults	Cycle*	Alloc.	Rnd.?	Alt.?	JIRA#
HDFS 2	1)	Lease recovery	IOE in IBR; IOE in write pipeline	1D | 2E | 0N	1		✓	17661
	2)	Edit log flushing	IOE in IBR after NN failover	1D | 1E | 0N	1	✓		17836
	3)	Block recovery	IOE in block recovery	1D | 1E | 0N	1	✓	✓	17662
	4)	Write pipeline	IOE in block recovery, IBR, and write pipeline	1D | 3E | 0N	2			17837
	5)	Block cache	IOE in write pipeline; DN timeout	1D | 1E | 1N	2			17660
	6)	IBR	IOE in IBR	1D | 1E | 0N	3			17780
HDFS 3	1)	Block deletion	IOE in write pipeline; DN timeout	1D | 1E | 1N	2	✓		17838
	2)	Block reconstruction; IBR	DN timeout; IOE in replication	2D | 1E | 1N	3			17782
HBase	1)	Write ahead log (WAL)	Premature `EndOfFile` in WAL	1D | 0E | 1N	1	✓	✓	29600
	2)	Region assignment	IOE in assignment RPC; Node exclusion	1D | 1E | 1N	3			29006
Flink	1)	Task worker	Head task failure; Sink task cancellation	1D | 2E | 0N	1	✓		38367
	2)	Aggregation task	Task state transition failure; Barrier task failure	1D | 2E | 0N	2	✓		38368
OZone	1)	Container report queue	Event queue dispatch failure	1D | 0E | 1N	1			13020
	2)	Heartbeat handling	IOE in pipeline construction; Pipeline unhealthy	1D | 1E | 1N	2	✓	✓	11856_1
	3)	Replication command handling	IOE in replication and pipeline construction	1D | 2E | 0N	3			11856_2

*D: Delay; E: Exception; N: Negation

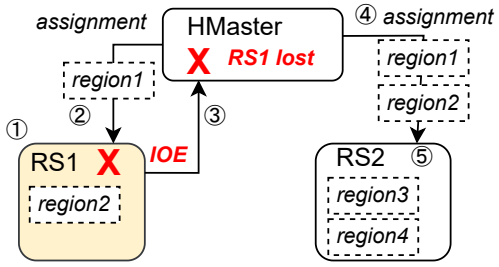

Figure 7. The case study. "RS" stands for RegionServer, and "IOE" stands for IOException. The yellow box indicates a delay.

fuzzer, widely being used in the security research community [72, 92] for distributed system testing.

We apply Jepsen on Flink [13] and Blockade on OZone [15] with their existing test cases. Due to the lack of Jepsen or Blockade integration and test workload, HBase and both versions of HDFS are not included in this comparison.

The result shows that none of the self-sustaining cascading failures detected by CSnake can be detected by fuzzing-based techniques. This highlights the necessity of causal stitching and fine-grained fault injection in CSnake for detecting self-sustaining cascading failures.

8.3 Case Study

In this section, we present case studies on two of the confirmed self-sustaining cascading failures.

8.3.1 Region Deployment Retry in HBase.

The second HBase bug in Table 3 is a self-sustaining cascading failure similar to the AWS motivating example in §1. HBase is a distributed database system that divides tables into regions managed by multiple RegionServers (RS) under master nodes.

At high level, RSes in a heavily loaded cluster may throw IOExceptions (IOEs) when handling RPC requests. The self-sustaining cascading failure happens in a cluster with a lot of write and table creation requests, which place significant load on the RSes managing these tables (① in Figure 7). Some region assignment operations on these RSes may time out and throw IOEs (②). When an RS throws an IOE, it is excluded from the master node's `FavoredStochasticBalancer`, which requires at least three live RSes to function properly. A reduced number of live RSes causes the load balancer to fail (③). An improper handling logic blindly retries the assignment indefinitely (④), further increasing the load on the RSes (⑤). This creates a self-sustaining cycle, leaving the cluster unable to process additional region assignment requests.

To trigger this bug in a testing environment with a single-fault (i.e., delay) injection, we need a workload with the following conditions:

1. It contains many region assignment requests.
2. A cluster configuration prone to be overloaded (3 nodes).
3. Load balancer configured to `F.S.Balancer`.
4. A workload long enough to observe the cycle.

However, no single test in HDFS satisfies all these conditions (as shown in §8.2), and CSnake detects this cycle

by injecting one delay, one exception, and one return-value negation into **three separate tests** t_1, t_2, and t_3.

First, the delay is injected into the region deployment loop in t_1 with multiple table creation and clones (cf. ① in Figure 7). This overloads the cluster, causing IOEs in the region assignment request handler (cf. ②). This causal relationship is only detected in t_1, because other tests only exercise limited number of region assignments.

In the second experiment, the same IOE is injected into the region assignment request handler in a test (t_2) for RS fault tolerance, triggering a negated return value in the load balancer's status checker (cf. ③). This causal relationship is only detected in t_2, because t_2 is the only test that uses `FavoredStochasticBalancer` with a cluster of 3 nodes, and the error detector `canPlaceFavoredNodes()` in `FavoredStochasticBalancer` only runs into an error (negation) if the cluster has fewer than 3 nodes (e.g., t_3 uses `FavoredStochasticBalancer` but has 5 nodes).

Finally, in the third experiment (t_3), the return-value negation is injected into the load balancer during a test for the balancer itself, increasing the iterations of the region deployment loop (cf. ④ and ⑤), indicating an increase in the workload being handled that can potentially cause delays. This causal relationship is only observed in t_3, because it is the only test that uses `FavoredStochasticBalancer` configuration and has a long enough workload to observe the increased iterations (not in t_2 because it exits prematurely after the IOE).

8.3.2 Bypassed IBR Throttling in HDFS.

The sixth bug detected on HDFS 2 is a logic error that a failed incremental block report (IBR) is retried immediately at the next heartbeat, incorrectly ignoring the configured IBR interval. The unconstrained, immediate retry of IBR can cause a self-sustaining cascading failure if the original IBR failure is due to a server overload. This happens when the cluster is under high user traffic with many modifications, which triggers many large-sized IBRs.

CSnake detects this bug by linking two injections in two different tests t_1 and t_2: 1) injecting an IBR processing delay into a workload with over 5,000 blocks to test the load balancer, and 2) injecting an RPC exception for IBR into a workload testing the IBR report interval configuration.

In fault injection using t_1, the RPC exception is observed, but not an increase in IBR count. Because t_2 is the only one with IBR throttling configured. Without it, IBRs are sent with every heartbeat. As a result, although IBR failures occur in the first workload, they are still sent at the original frequency and cannot be detected by execution trace comparison.

In fault injection using t_2, the IBR increment is observed after the RPC exception, indicating a potential delay. However, t_2 involves only 8 file blocks compared to 5,000 in t_1, making IBR processing delays less likely to cause timeouts.

Table 4. Number of self-sustaining cascading failures reported by CSnake and the clustering results. Column "System" is the targeted distributed system. Columns "Cycle", "Cluster", and "TP" shows the number of cycles reported, distinct cycle clusters, and true positive clusters respectively. The numbers outside the parentheses are from a beam search with unlimited number of injected faults, while the numbers inside are from a beam search limiting to one delay injection.

System	Cycle	Cluster	TP	System	Cycle	Cluster	TP
HDFS 2	38 (23)	15 (9)	6 (6)	Flink	48 (27)	35 (17)	2 (2)
HDFS 3*	149 (59)	36 (14)	4 (3)	OZone	29 (17)	11 (7)	3 (3)
HBase	72 (26)	24 (7)	2 (2)	**Total***			**17 (16)**

*Including two duplicated clusters also detected in HDFS 2

8.4 Cycle Clustering and False Positive Rate

Table 4 shows the number of self-sustaining cascading failures, distinct failure clusters, and the true positive reported by CSnake on each system. The numbers outside the parentheses are from a beam search without an upper bound on the number of injected faults, while the numbers inside the parentheses are from a beam search for cycles with at most one delay injection but unlimited exceptions or negations.

Limiting the beam search to one delay injection generally reduces false positives while still identifying most failures. Users can adjust the number of injected faults to balance between accuracy and completeness.

8.4.1 Effectiveness of Cycle Clustering.

We manually inspect all the self-sustaining cascading failures and the *causally equivalent fault* clusters on HDFS 2 and confirm that the clustering algorithm effectively groups similar cycles together. Examples of the clustered cycles involve faults from 1) the write pipeline, 2) DataNode command generation and processing.

However, we do observe false positives in the clustering results due to 1) non-determinism and 2) insufficient test execution. For example, a fault cluster in a HDFS run mixed four correctly-clustered data-race faults with two false positives: a failed security check whose unique causal consequence would have been revealed with more test cases, and a logging error clustered due to non-determinism. This could cause missed self-sustaining cascading failures in detection if the false positive fault is selected during the 3PA allocation, though their impact is limited as long as the majority are clustered correctly.

The higher numbers of cycles and clusters reported in HDFS 3 are due to the extensive usage of asynchronous tasks and event queues, where errors in the issuer and the executor of asynchronous tasks are processed by different error handlers. Compared to synchronous execution, this increases the number of error handlers, resulting in a larger number of cycles and fault clusters.

8.4.2 False Positive Analysis. The false positive clusters reported by CSnake are mainly due to the following reasons:

1. In cycle searches with unlimited fault injections, many false positives (about 70% on HDFS 2) are due to "expected" contentions, such as between HDFS clients with heavy read and write operations. These cases require solutions such as throttling or capacity increases. We do not consider them as true positives.
2. Increased loop iterations don't always indicate delays, especially if throttling mechanisms are already in place. These mechanisms are often implemented ad-hoc, making them difficult for static analyzers to identify.

From our experience, the false positives due to the above reasons can be filtered out manually relatively easily, given adequate understanding of the system under test.

Theoretically, CSnake could still report a self-sustaining cascading failure involving fault propagation chains from tests with contradictory conditions after our local compatibility check. However, we do not observe such cases in the clusters reported by CSnake.

8.5 Performance Overhead

CSnake's instrumentation and monitoring introduces an average of 185% runtime overhead in the profile run, ranging from 63% to 376%. This overhead is primarily caused by branch tracing and call stack recording, and can be further reduced using hardware-based tracing such as IntelPT [46]. While sampling can be used to reduce the runtime overhead, we opt not to implement sampling in branch tracing to maximize the detection accuracy as CSnake is intended to be deployed in the testing environment instead of the production environment.

8.6 Effort of Applying CSnake to a New System

CSnake is designed to be system-agonistic and requires as little input from users as possible. Applying CSnake to a new system only requires two steps: 1) compiling the target system, and 2) setting up the test environment with system dependencies. Two graduate students who have no knowledge of the tool can independently apply CSnake to a new system within half a day without consulting the authors.

9 Discussion

In this section, we discuss CSnake's generality, its usage in real-world scenarios, and lessons learned from the self-sustaining cascading failures detected.

9.1 Generality of CSnake

Across distributed system components. CSnake can identify causal relationships across components of distributed systems (currently supports JVM-based components). For example, CSnake can identify causal relationships between faults in HBase and its underlying HDFS, provided that integration tests driving multiple system components are available.

Generality of faults. CSnake's fine-grained fault injection (§4.1) is both representative and general. First, we achieve completeness in exception injection by targeting all `throw` statements, which are then conservatively pruned via a rule-based approach. Second, we also achieve completeness in contention injection for self-sustaining cascading failures by simulating their common cause – workload-induced contention [48, 78] – with spinning delay. Third, we make a practical tradeoff with the system-specific error injection by using a heuristic (i.e., targeting boolean-returning functions) to maintain generality. While this heuristic may miss some error detectors implemented as branch conditions, such cases are often captured by exception injections if they guard `throw` statements. Conversely, a fully complete approach would require input from developers to distinguish normal branch conditions from the remaining system-specific error detectors.

9.2 Using CSnake in Real-World Scenarios

We expect CSnake to be applied before releases to catch potential self-sustaining cascading failures due to its exhaustive testing approach. To adapt CSnake for regular regression testing, CSnake's budget allocation algorithm can be refined to focus on faults in the classes or packages that 1) have been involved in prior failures or 2) involve heavy code changes and newly introduced test workloads. However, this remains a future direction that requires thorough evaluation.

9.3 Lessons Learned

Several ways can be applied to reduce the possibility of encountering self-sustaining cascading failures in real-world distributed systems. First, workload should never be piggybacked on critical requests (e.g., heartbeat or failure detectors). A surge in the user traffic can easily drive the system into a cascading failure. Second, distributed systems should properly implement throttling mechanisms by proactively monitoring current system load such as the queue length. In addition to dropping requests at the server side, clients should be properly informed to refrain themselves from retrying to avoid a retry storm. Third, asynchronous requests can be used with priority queues to ensure that critical operations are handled first in a unusual traffic spike.

10 Related Work

Cascading Failure. CSnake gains insight from the work of Qian et al. [78] and Huang et al. [48] on self-sustaining cascading failures in distributed systems. Following their work, modeling [42], diagnosis [45, 62, 80], and mitigation [67, 73] techniques have been proposed and studied. However, none

of them focuses on exposing self-sustaining cascading failures caused using testing or fault injection. Li et al. [64] proposes a technique to detect *performance cascading failures*, a subtype of cascading failures that only involves performance interference by tracking a variant of happens-before relationship causality. In comparison, CSnake detects fault propagation involving both performance interference and functional interference through counterfactual causality analysis.

Automatic Bug Detection and Diagnosis in Distributed Systems. Recently, there has been a rise in the focus on automatically detecting and diagnosing failures in distributed systems, including performance cascading bugs [64], exception-dependent failures [62], upgrade failures [90], timeout bugs [30], and partial failures [86]. Compared with their work, CSnake's causal stitching links multiple faults injected in to the system. None of the prior work can provide detailed insight into the root causes as well as the propagation chains of the self-sustaining cascading failures.

Fault Injection. Traditionally, fault injection techniques [23, 24, 47, 70, 75] focus on single-node system. Recently, it has become a popular [9, 19, 20, 29, 31, 36, 40, 56, 57, 63, 68, 70, 71, 74, 83, 89] technique in testing distributed systems. Compared with CSnake, many of them inject coarse-grained external faults such as node failures and network partitions to expose crash recovery bugs. Although more advanced techniques exist, such as injecting faults during variable accesses [68], none of them are able to detect and expose the entire causal chain of self-sustaining cascading failures. CSnake performs fine-grained injection of delay, exception, and return value negation into the system. Combined with the fault causality analysis, CSnake is the only tool that can detect and diagnose self-sustaining cascading failures in distributed systems.

Fuzzing. Fuzzing is a software testing technique that feeds programs with large volumes of random inputs to uncover bugs with predefined oracles such as crashes. Prior work mainly applies fuzzing techniques to single-node systems and programs [2, 21, 22, 26, 33, 35, 37, 51–53, 87], such as operating systems [39, 50, 82, 85], network protocols [77, 93], and filesystems [52, 59]. Recently, fuzzing has also been applied to test distributed systems [38, 72, 92] and shows promising results.

CSnake complements existing black-/grey-box fuzzing techniques. While fuzzing explores new behaviors by mutating test inputs (e.g., message sequences, code coverage), CSnake extends fault causality chains, which can be used as feedback information for existing fuzzers. For instance, when CSnake stitches $f_1 \rightarrow f_2$ in t_1 and $f_2 \rightarrow f_3$ in t_2 to $f_1 \rightarrow f_2 \rightarrow f_3$, such a causality chain can guide the fuzzers to mutate t_1 and t_2 together (e.g., merging workload in both tests), helping triggering self-sustaining cascading failures.

11 Conclusion

We propose CSnake, a fault-injection-based testing and detection framework for self-sustaining cascading failures in distributed systems. CSnake leverages the novel idea of causal stitching to expose the complex fault propagation chains and detects potential self-sustaining cascading failures due to incorrect program logic. We evaluate CSnake on five popular distributed systems and find 15 new self-sustaining cascading failures, five of which have been confirmed with two fixed by developers. CSnake is effective with little burden on targeted users.

Acknowledgments

The authors thank Jia-Ju Bai, our shepherd, and anonymous reviewers for their constructive comments. This research is partially supported by NSF 1901242, 2006688, and 2300562. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

References

[1] Summary of the AWS Service Event in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/073024/ [Online; accessed 2025-05-10].

[2] american fuzzy lop. https://github.com/google/AFL.

[3] Apache Flink. https://flink.apache.org/.

[4] Apache HBase. https://hbase.apache.org/.

[5] Apache OZone. https://ozone.apache.org/.

[6] async-profiler: Sampling CPU and HEAP profiler for Java featuring AsyncGetCallTrace + perf_events. https://github.com/async-profiler/async-profiler.

[7] Blockade. https://github.com/worstcase/blockade.

[8] CRIU. https://criu.org/Main_Page.

[9] Distributed Systems Safety Research. https://jepsen.io/.

[10] HDFS architecture guide. https://hadoop.apache.org/docs/r2.10.2/.

[11] HDFS architecture guide. https://hadoop.apache.org/docs/r3.4.1/.

[12] HDFS High Availability. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html.

[13] Jepsen Flink. https://github.com/apache/flink/tree/release-1.14.6/flink-jepsen.

[14] JUnit 4. https://junit.org/junit4/.

[15] OZone Blockade Tests. https://github.com/apache/ozone/tree/ozone-1.4.0/hadoop-ozone/fault-injection-test/network-tests/src/test/blockade.

[16] Parameterized tests. https://github.com/junit-team/junit4/wiki/Parameterized-tests.

[17] Simplify Java tracing, monitoring and testing with Byteman. https://byteman.jboss.org/.

[18] T.J. Watson Libraries for Analysis. https://github.com/wala/WALA.

[19] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of Network-Partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 51–68.

[20] Peter Alvaro, Joshua Rosen, and Joseph M Hellerstein. 2015. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 331–346.

[21] Jia-Ju Bai, Zi-Xuan Fu, Kai-Tao Xie, and Zu-Ming Jiang. 2023. Testing error handling code with software fault injection and error-coverage-guided fuzzing. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2023), 1724–1739.

[22] Jia-Ju Bai, Hao-Xuan Song, and Shi-Min Hu. 2024. Multi-dimensional and message-guided fuzzing for robotic programs in robot operating system. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 763–778.

[23] Jia-Ju Bai, Yu-Ping Wang, Hu-Qiu Liu, and Shi-Min Hu. 2016. Mining and checking paired functions in device drivers using characteristic fault injection. *Information and Software Technology* 73 (2016), 122–133.

[24] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. 2016. Testing error handling code in device drivers using characteristic fault injection. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 635–647.

[25] Joeran Beel, Bela Gipp, Stefan Langer, and Corinna Breitinger. 2016. Paper recommender systems: a literature survey. *International Journal on Digital Libraries* 17, 4 (2016), 305–338.

[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[27] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[28] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.

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

[30] Yuanliang Chen, Fuchen Ma, Yuanhang Zhou, Ming Gu, Qing Liao, and Yu Jiang. 2024. Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1939–1955.

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

[32] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)* 30, 1 (2012), 1–49.

[33] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[34] Anna Fariha, Suman Nath, and Alexandra Meliou. 2020. Causality-guided adaptive interventional debugging. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 431–446.

[35] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[36] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults. *ACM Transactions on Storage (TOS)* 13, 3 (2017), 1–33.

[37] Si-Miao Gao, Pengcheng Wang, Jia-Ju Bai, Jia-Wei Yu, and Haizhou Wang. 2025. Detecting Lifecycle-Related Concurrency Bugs in ROS Programs via Coverage-Guided Fuzzing. *IEEE Transactions on Information Forensics and Security* (2025).

[38] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2211–2223.

[39] Sishuai Gong, Wang Rui, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2025. Snowplow: Effective kernel fuzzing with a learned white-box test mutator. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 1124–1138.

[40] 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. 2011. FATE and DESTINI: A framework for cloud recovery testing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*.

[41] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. 2013. Failure recovery: When the cure is worse than the disease. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.

[42] Farzad Habibi, Tania Lorido-Botran, Ahmad Showail, Daniel C Sturman, and Faisal Nawab. 2024. MSF-model: Queuing-based analysis and prediction of metastable failures in replicated storage systems. In *2024 43rd International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 12–22.

[43] Seungjae Han, Kang G Shin, and Harold A Rosenberg. 1995. Doctor: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium*. IEEE, 204–213.

[44] David Harris. AWS Outage Hits Amazon Services, Ring, Whole Foods, Alexa. https://www.crn.com/news/cloud/2024/aws-outage-hits-amazon-services-ring-whole-foods-alexa [Online; accessed 2025-05-10].

[45] Vipul Harsh, Wenxuan Zhou, Sachin Ashok, Radhika Niranjan Mysore, Brighten Godfrey, and Sujata Banerjee. 2023. Murphy: Performance diagnosis of distributed cloud applications. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 438–451.

[46] Jack Henschel. Intel Processor Tracing. https://blog.cubieserver.de/publications/Henschel_Intel-PT_2017.pdf.

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

[48] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 73–90.

[49] Minseok Jeon and Hakjoo Oh. 2022. Return of CFA: call-site sensitivity can be superior to object sensitivity even for object-oriented programs. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

[50] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. 2019. Fuzzing error handling code in device drivers based on software fault injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 128–138.

[51] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *29th USENIX Security Symposium (USENIX Security 20)*. 2595–2612.

[52] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*.

[53] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful fuzzing for database management systems with complex and valid SQL query generation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4949–4965.

[54] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 77–88. doi:10.1145/2254064.2254075

[55] Brittany Johnson, Yuriy Brun, and Alexandra Meliou. 2020. Causal testing: understanding defects' root causes. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering*. 87–99.

[56] Pallavi Joshi, Haryadi S Gunawi, and Koushik Sen. 2011. PREFAIL: A programmable tool for multiple-failure injection. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 171–188.

[57] Xiaoen Ju, Livio Soares, Kang G Shin, Kyung Dong Ryu, and Dilma Da Silva. 2013. On fault resilience of openstack. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–16.

[58] Leonard Kaufman and Peter J Rousseeuw. 2009. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons.

[59] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 147–161.

[60] Daniel L Kiskis and Kang G Shin. 1996. A synthetic workload for a distributed real-time system. *Real-Time Systems* 11, 1 (1996), 5–18.

[61] Antje Krause, Jens Stoye, and Martin Vingron. 2005. Large scale hierarchical clustering of protein sequences. *BMC bioinformatics* 6 (2005), 1–12.

[62] Ao Li, Shan Lu, Suman Nath, Rohan Padhye, and Vyas Sekar. 2024. ExChain: Exception Dependency Analysis for Root Cause Diagnosis. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 2047–2062.

[63] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. 2019. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 162–180.

[64] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[65] Wang Li, Zhouyang Jia, Shanshan Li, Yuanliang Zhang, Teng Wang, Erci Xu, Ji Wang, and Xiangke Liao. 2021. Challenges and opportunities: an in-depth empirical study on configuration error injection testing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 478–490.

[66] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A principled approach to selective context sensitivity for pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–40.

[67] Yueying Li, Daochen Zha, Tianjun Zhang, G. Edward Suh, Christina Delimitrou, and Francis Y. Yan. 2023. Mitigating Metastable Failures in Distributed Systems with Offline Reinforcement Learning. In *The First Tiny Papers Track at ICLR 2023, Tiny Papers @ ICLR 2023, Kigali, Rwanda, May 5, 2023*. https://openreview.net/forum?id=zYF6NLJl6LM

[68] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Yang, and Liang You. 2019. Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 114–130.

[69] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. 2009. *Introduction to information retrieval*. Cambridge University Press.

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

[71] Christopher S Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-level fault injection testing. In *Proceedings of the ACM Symposium on Cloud Computing*. 388–402.

[72] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. 2023. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1615–1629.

[73] Justin J Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, et al. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 607–622.

[74] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 33–50.

[75] Roberto Natella, Domenico Cotroneo, and Henrique S Madeira. 2016. Assessing dependability with software fault injection: A survey. *ACM Computing Surveys (CSUR)* 48, 3 (2016), 1–55.

[76] Judea Pearl and Dana Mackenzie. 2018. *The book of why: the new science of cause and effect*. Basic books.

[77] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[78] Shangshu Qian, Wen Fan, Lin Tan, and Yongle Zhang. 2023. Vicious Cycles in Distributed Software Systems. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 91–103.

[79] Olin Shivers. 1988. Control flow analysis in Scheme. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 164–174.

[80] PC Sruthi, Zinan Guo, Deming Chu, Zhengyan Chen, and Yongle Zhang. 2024. Demystifying the Fight Against Complexity: A Comprehensive Study of Live Debugging Activities in Production Cloud Systems. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 341–360.

[81] Student. 1908. The probable error of a mean. *Biometrika* (1908), 1–25.

[82] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358.

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

[84] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 278–291.

[85] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer.

[86] Haoze Wu, Jia Pan, and Peng Huang. 2024. Efficient Exposure of Partial Failure Bugs in Distributed Systems with Inferred Abstract States. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1267–1283.

[87] Kai-Tao Xie, Jia-Ju Bai, Yong-Hao Zou, and Yu-Ping Wang. 2022. ROZZ: property-based fuzzing for robotic programs in ROS. In *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 6786–6792.

[88] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*.

[89] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 2021. 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*. 458–472.

[90] Yongle Zhang, Junwen Yang, Zhuqi Jin, Utsav Sethi, Kirk Rodrigues, Shan Lu, and Ding Yuan. 2021. Understanding and detecting software upgrade failures in distributed systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 116–131.

[91] Zhongheng Zhang, Fionn Murtagh, Sven Van Poucke, Su Lin, and Peng Lan. 2017. Hierarchical cluster analysis in clinical research with heterogeneous study population: highlighting its visualization with R. *Annals of translational medicine* 5, 4 (2017), 75.

[92] Yonghao Zou, Jia-Ju Bai, Zu-Ming Jiang, Ming Zhao, and Diyu Zhou. 2025. Blackbox Fuzzing of Distributed Systems with Multi-Dimensional Inputs and Symmetry-Based Feedback Pruning. Network and Distributed System Security (NDSS) Symposium 2025.

[93] Yong-Hao Zou, Jia-Ju Bai, Jielong Zhou, Jianfeng Tan, Chenggang Qin, and Shi-Min Hu. 2021. TCP-Fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 489–502.

A Details on 3PA Protocol

Suppose there are n possible fault locations in the system, namely $f_1, f_2, ..., f_n$, making up the fault space $\mathbb{F}$. Injecting f_i into test t_j causes a m additional faults:

$$I(f_i, t_j) = [f_{k_1}, f_{k_2}, ..., f_{k_m}] \quad k_x \in [1..n], x \in [1..m] \tag{1}$$

A.1 IDF Vectorization

Each $I(f_i, t_{i_1})$ is vectorized using the inverse document frequency (IDF) [69], resulting in a real vector of length n, with each element ranging from 0 to 1. Vectorization is a common technique before applying clustering algorithms [69].

$$V(f_i, t_j) = \text{IDF}_\text{v}(I(f_i, t_j), \mathbb{F}) \quad \in [0, 1]^n \tag{2}$$

IDF measures how often a fault is triggered in all the fault injection runs. Formally, it is defined [69] as:

$$\text{IDF}(f, \mathbb{F}) = \log \frac{1 + N}{1 + N_f} \tag{3}$$

N is the total number of fault injection experiments, and N_f is the number of experiments that triggers the additional fault f. The plus one smooths the IDF values and prevents zero divisions.

To vectorize $I(f_i, t_j)$ using IDF, each triggered fault is replaced with its IDF value, and other elements in $V(f_i, t_j)$ are set to zero. L2 normalization is applied to the final vector. Formally, the vectorization is defined as:

$$V(f_i, t_j) = \text{IDF}_\text{v}(I(f_i, t_j), \mathbb{F}) = \frac{(v_1, v_2, ..., v_n)}{\|(v_1, v_2, ..., v_n)\|}$$
$$v_x = \begin{cases} \text{IDF}(f_x, \mathbb{F}), & \text{if } f_x \in I(f_i, t_j) \\ 0, & \text{otherwise} \end{cases} \tag{4}$$
$$\text{where} \quad x \in [1..n]$$

A.2 Phase One

In phase one of the 3PA protocol, CSnake injects each fault f_i into workload t_{i_1} that reaches f_i's program location and has the highest code coverage. We perform a hierarchical clustering [58] of the faults in $\mathbb{F}$ using the cosine distance D_c between all vectorized interferences $V(F_i, T_{i_1})$, as defined below:

$$\text{D}_\text{c}(V(f_i, t_{i_1}), V(f_j, Tt_{j_1})) = 1 - \frac{V(f_i, t_{i_1}) \cdot V(f_j, t_{j_1})}{\|V(f_i, t_{i_1})\| \|V(f_j, t_{j_1})\|} \tag{5}$$
$$\text{where} \quad i, j \in [1..n], i \neq j$$

By the end of phase one, each fault f_i is clustered in to a group G_j with other faults having similar interferences on the system once injected.

A.3 Phase Two

In phase two of the 3PA protocol, an intra-cluster interference similarity score is calculated for each cluster G_i. This score is the average pairwise cosine distance of all vectorized interference results within the cluster. That is, suppose G_i has p faults $f_{i_1}, f_{i_2}, ..., f_{i_p}$, f_{i_k} is injected into q_k different workloads $t_1^{i_k}, t_2^{i_k}, ..., t_{q_k}^{i_k}$, cluster G_i has $\Sigma_{k=1}^{p} q_k$ vectorized interference results $V(f_i, t_j)$, the similarity score is defined as:

$$\text{SimScore}(G_i) = 1 - \overline{\text{D}_\text{c}(V(f_a, t_x^a), V(f_b, t_y^b)} \quad \in [0, 1]$$
$$\text{where} \quad a, b \in [1..p], x \in [1..q_a], y \in [1..q_b], a \neq b \tag{6}$$

$\text{SimScore}(G_i)$ ranges from 0 to 1. A value of 1 indicates that all faults in cluster G_i triggers the same set of additional faults among all injection runs.

A.4 Phase Three

In phase three of the 3PA protocol, the budget allocation weight for cluster G_i is defined as:

$$\text{W}(G_i) = \max(\epsilon, 1 - \text{SimScore}(G_i)) \quad \in [0, 1] \tag{7}$$

Each group has a minimum weight ϵ of 0.01, ensuring every cluster receives some budget, even with perfectly matched intra-group interference results.

B Additional Implementation Details

B.1 Dynamic Call Graph Collection

Loop scalability analysis in CSnake (§4.1) requires a call graph to identify the functions reachable from each loop. During development, we find that WALA's static call graph struggles with polymorphism. More accurate algorithms such as 2-CFA [79] do not scale well for large systems, such as HDFS with over 359,000 lines of code. To address this, we uses `async-profiler` [6]'s CPU sampler alongside CSnake's tracing capabilities to collect runtime stack snapshots, from which a dynamic call graph is reconstructed.

B.2 Parameterized JUnit Test

A challenge in test execution is the wide use of parameterized JUnit tests [16], which reuses the test methods with different input parameters, often generated dynamically. CSnake uses a modified version of JUnit that skips the test body execution and focuses solely on parameter generation. A custom test filter intercepts this process and captures parameters for testing.

C Artifact Appendix

This artifact contains a minimum working example of the CSnake.

C.1 Description & Requirements

C.1.1 How to access. The code can be downloaded at https://doi.org/10.5281/zenodo.17049891.

The Zenodo artifact contains one file, `CSnake-AE.tar.zst`, it contains a folder `CSnake-AE` with a `README.md` inside. The readme file contains all the necessary steps for the artifact evaluation process.

It is mandatory that the file is downloaded into "/" under Linux and extracted there. In other words, the code will be located at /CSnake-AE.

C.1.2 Hardware dependencies. Detailed requirements are listed inside README.md. We recommend using a machine with at least 256GB of memory and 50GB of SSD storage space.

C.1.3 Software dependencies. We list detailed commands for installing all the software dependencies in README.md.

C.1.4 Benchmarks. None.

C.2 Set-up

See README.md for details.

C.3 Evaluation workflow

README.md contains nine steps of demonstrating the functionality of CSnake.

CSnake's static analyzer is executed in Step 1. CSnake's profile run of integration tests is executed in Step 2–4. Step 5 analyzes the output and prepare CSnake for fault injection runs. Step 6 and 7 executes the fault injection run. Meanwhile, our runtime agents for fault injection and monitoring is exercised. Step 8 performs the fault causality analysis. Step 9 runs the bug detector, performing the local compatibility check and parallel beam search.