Fail-Slow Hardware Failure Bug Analysis and Detection for Cloud Systems

GEN DONG and YU HUA, Huazhong University of Science and Technology, China
YONGLE ZHANG, Purdue University, USA
ZHANGYU CHEN and MENGLEI CHEN, Huazhong University of Science and Technology, China

Fail-slow hardwares are still running and functional, but in a degraded mode, thus slower than their expected performance. Bugs triggered by fail-slow hardwares cause severe cloud system failures. Existing testing tools fail to efficiently detect these bugs due to overlooking their characteristics. In order to address this problem, this paper provides a bug study that analyzes 48 real-world fail-slow hardware failures from typical cloud systems. We observe that (1) fail-slow hardwares make high-level software components vulnerable, including synchronized and timeout mechanisms; (2) the fine granularity of fail-slow hardwares is necessary to trigger these bugs. Based on these two observations, we propose Sieve, a fault injection testing framework for detecting fail-slow hardware failure bugs. Sieve statically analyzes target system codes to identify synchronized and timeout-protected I/O operations as candidate fault points and instruments hooks before candidate fault points to enable fail-slow hardware injection. To efficiently explore candidate fault points, Sieve adopts grouping and context-sensitive injection strategies. We have applied Sieve to three widely deployed cloud systems, i.e., ZooKeeper, Kafka, and HDFS. Sieve has detected six unknown bugs, two of which have been confirmed.

CCS Concepts: • **Software and its engineering → Correctness**; • **Computer systems organization → Reliability**;

Additional Key Words and Phrases: Cloud systems, fail-slow hardware, fault injection, bug detection

ACM Reference Format:
Gen Dong, Yu Hua, Yongle Zhang, Zhangyu Chen, and Menglei Chen. 2025. Fail-Slow Hardware Failure Bug Analysis and Detection for Cloud Systems. *ACM Trans. Comput. Syst.* 37, 4, Article 111 (August 2025), 34 pages.
https://doi.org/XXXXXXX.XXXXXXX

This paper, originally titled "Understanding and Detecting Fail-Slow Hardware Failure Bugs in Cloud Systems" [15], was invited by ACM TOCS for extended publication through recommendation by the Chairs of ATC 2025. The additional contents include: (1) support for exception injection to simulate exception propagation caused by fail-slow hardware; (2) a data corruption checker and an optional agent-based failure validation component; (3) additional implementation details on timeout-value collection and memory access tracing; and (4) an extended evaluation with checker-level bug detection results, agent-based validation results, and further discussion of limitations and extensibility.

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022.

Authors' Contact Information: Gen Dong, Huazhong University of Science and Technology, WuHan, HuBei, China, gdong@hust.edu.cn; Yu Hua (Corresponding author), Huazhong University of Science and Technology, WuHan, HuBei, China, csyhua@hust.edu.cn; Yongle Zhang, Purdue University, West Lafayette, Indiana, USA, yonglezh@purdue.edu; Zhangyu Chen, Huazhong University of Science and Technology, WuHan, HuBei, China, cs.zychen@gmail.com; Menglei Chen, Huazhong University of Science and Technology, WuHan, HuBei, China, chenml@hust.edu.cn.

ACM 1557-7333/2025/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

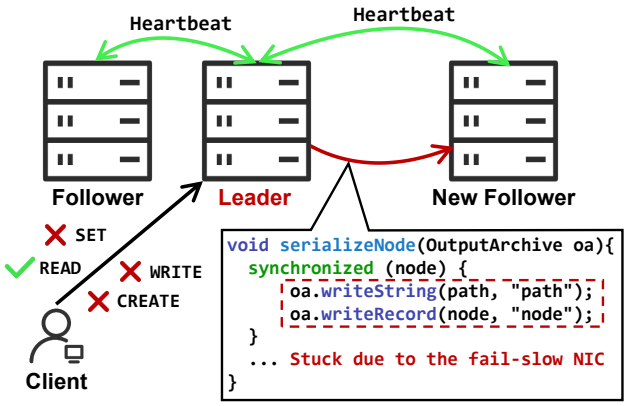

Fig. 1. A production ZooKeeper failure due to the fail-slow NIC [13]. The writeRecord operation becomes slow within the synchronized block. The update-type requests are blocked. Developers fixed the bug by copying the node object and serializing the copy out of the synchronization.

1 Introduction

Cloud systems have become the backbone of numerous modern applications [34, 63, 67, 71]. However, fault-triggered failures in cloud systems cause severe loss of customer satisfaction and revenues for service providers [11, 22, 59]. To achieve high reliability, cloud systems need to correctly handle various faults [1, 12, 42, 47]. Based on the scope of faults, we classify them into two categories: **coarse-grained** and **fine-grained** faults. Specifically, when only considering I/O operations in a cloud system, let D=$\{D_1,D_2,...,D_n\}$ represent the set of disk I/O operations and N=$\{N_1,N_2,...,N_m\}$ represent the set of network I/O operations. Coarse-grained faults affect all associated I/O operations, e.g., the I/O operations in D fail in fail-stop disks [51] and those in $D \cup N$ fail in node crashes [19, 47]. Fine-grained faults affect a subset of associated I/O operations, e.g., fail-slow Network Interface Cards (NICs) [25] slow down N_{sub} ($N_{sub} \subset N$). Coarse-grained faults have been well studied and handled by existing well-designed fault tolerance mechanisms [5, 21, 58, 62]. However, emerging fine-grained faults propose new challenges to the reliability of cloud systems.

Recently, fail-slow hardwares [25, 49] received an increasing amount of attention, which deliver lower-than-expected performance and cause fine-grained faults. For example, the throughput of a 1-Gbps NIC might suddenly drop by several orders of magnitude to 1 Kbps due to partial buffer corruption and retransmission [14], which slows down partial network I/O operations. Lu et al. [49] indicate that the fail-slow NVMe SSD has become a widespread and severe problem. On average, 1.41% of NVMe SSDs (over one million NVMe SSDs) are infected within four months of monitoring. Moreover, fail-slow hardware incidents always consume hours or even months to detect. Only 1% of the cases are detected in minutes [25]. As a consequence, once fail-slow hardwares bring cloud systems down, it requires large human effort to detect the root cause. For example, several AWS services become unavailable due to unexpected high network latency [59], which takes developers nine hours to detect the root cause. Figure 1 presents another real-world fail-slow hardware failure from ZooKeeper [13] which is a widely used distributed coordination to tolerate leader and follower crashes. Specifically, an entire cluster becomes a near-freeze status. `write()` and `create()` requests fail, but `read()` requests can be successfully processed. The developers at PagerDuty [56] spend five months to diagnose this failure. In this paper, we refer to the fault caused by the fail-slow

hardware as the **FSH**[1] **fault** and the bug triggered by the FSH fault as the **FSH bug**. We define ***fail-slow hardware failures*** as the failures caused by FSH bugs and refer to these failures as **FSH failures**.

To prevent the FSH fault from breaking cloud systems at runtime, it is essential to develop an FSH fault injection testing framework. Fault injection testing [1, 6, 17, 19, 24, 36, 37, 47, 53] is a commonly used technique for detecting fault-triggered bugs. There are enormous effort in developing various fault injection testing frameworks. However, existing schemes fail to efficiently detect FSH bugs due to overlooking the characteristics of FSH failures. To address this problem, we conduct a study of 48 real-world FSH failure cases from five large-scale and open-source systems. Furthermore, we observe that there are two key characteristics in efficiently detecting FSH failures like the example in Figure 1.

1) The vulnerable synchronized and timeout mechanisms cause all studied FSH failures. For example, synchronized tasks slowed down by fail-slow hardwares block other tasks for an uncertain time; data race can occur between the timeout task caused by the fail-slow hardware and the corresponding timeout handler with simple timing constraints. This characteristic can significantly prune the huge fault injection space. However, existing bug studies overlook this characteristic since they focus on either the fail-slow software or hardware. Unlike them, our bug study comprehensively analyzes how the fail-slow hardware affects the software.

2) The fine granularity of FSH faults is necessary to trigger FSH failures. One of the main reasons is that fine-grained FSH faults are relatively easy to escape the detection of internal checkers. For example, as shown in Figure 1, there are two network I/O operations, i.e., `serializeNode` and `Heartbeat`. The fail-slow NIC only slows down `serializeNode` so that the heartbeat thread in the leader can maintain the liveness of the cluster. Hence, the entire cluster becomes a *near-freeze status*. In this scenario, the fail-slow NIC escapes the detection of the heartbeat threads in the followers. The fine-grained fault injection is necessary to trigger such failure. Specifically, if a fail-stop NIC is injected in the leader, these two network I/O operations are affected. Subsequently, the heartbeat threads in the followers will detect this coarse-grained fault and start a new leader election, which drives the cluster into a healthy status. Most existing fault injection testing frameworks focus on coarse-grained faults. Although these schemes are enough to detect crash recovery bugs or distributed protocol bugs, they cannot effectively detect FSH bugs.

Inspired by these two key characteristics, we propose Sieve, a fault injection testing framework designed to detect FSH bugs in cloud systems. Sieve treats synchronized and timeout-protected I/O operations as candidate fault points. To identify candidate fault points, we investigate various synchronized and timeout mechanisms in cloud systems and conclude four general patterns from them. Based on these four patterns, Sieve identifies I/O operations in synchronized and timeout-protected domains through lightweight static analysis at the code instruction level. Besides, Sieve automatically instruments hooks before candidate fault points to precisely control the fine-grained FSH fault injection.

To efficiently explore these candidate fault points, Sieve adopts grouping and context-sensitive injection strategies. Specifically, Sieve groups fault points within the same basic block[2]. Because fault points within the same basic block are always under the same high-level system states (e.g., leader election and snapshot synchronization) and protected by the same fault-tolerance handler, which indicates these fault points trigger similar scenarios. Besides, Sieve leverages the context-sensitive injection strategy to (1) avoid exploring the same contexts of fault points and (2) find buggy contexts of the same fault points. During testing, the hook sends the fault injection query to

[1]FSH denotes fail-slow hardware.

[2]A sequence of consecutive instructions without any branch instructions.

Table 1. The numbers of FSH Failures.

ZooKeeper	HDFS	HBase	MapReduce	Cassandra
11	18	10	4	5

the injection controller. The controller checks the runtime context of the fault point, e.g., call stack and thread ID, and decides whether to inject the fault.

Fail-slow hardware manifests in diverse fault behaviors at runtime. To support such behaviors, Sieve integrates two fault injection modes. Specifically, Sieve implements (i) delay injection to model degraded performance of fail-slow hardware, and (ii) exception injection to simulate exception propagation caused by fail-slow hardware. These injection modes enable Sieve to cover a wide spectrum of FSH faults observed in practice. Moreover, Sieve implements three checkers: the log error checker, the gray failure checker and the data corruption checker, to detect various failure symptoms of FSH failures, e.g., node service unavailable, data loss and inconsistency. Additionally, to reduce the manual effort required for validating FSH failures, Sieve implements an optional agent-based failure validation component.

We have implemented a prototype of Sieve and evaluated it on three large-scale cloud systems, i.e., ZooKeeper [76], Kafka [38], and HDFS [2]. Moreover, Sieve has detected six unknown bugs in these systems. Besides, we have released Sieve (including bug study, scripts for reproducibility, and source codes) for public use at https://github.com/RabbitDong-on/Sieve.

In summary, this paper makes the following contributions:

- We conduct a study of 48 real-world FSH failure cases from five cloud systems. Our bug study helps developers to comprehensively understand the characteristics of FSH failures.
- We propose Sieve, a novel approach that enables fine-grained FSH fault injection for detecting FSH bugs. Based on our bug study, Sieve analyzes synchronized and timeout-protected I/O operations as candidate fault points. To efficiently explore these candidate fault points, Sieve adopts grouping and context-sensitive injection strategies.
- We extend Sieve to support diverse fail-slow hardware behaviors by integrating two fault injection modes, and to incorporate complementary failure checkers, with an agent-based component for validating reported FSH failures.
- We implement a prototype of Sieve and evaluate it on three large-scale real-world cloud systems. Sieve has detected six unknown bugs, two of which have been confirmed.

2 Methodology

To understand the FSH failure, we studied 48 real-world failures in five popular cloud systems shown in Table 1. We leverage the search tool in JIRA [60] to identify reports related to fail-slow hardwares. First, we search reports using the following keywords: “slow disk”, “slow storage”, “slow network”, “slow NIC”, and “slow switch”. Second, we exclude the issues that have priority of “Minor”, “Trivial” and “Low”. Third, we identify the issues that are indeed related to fail-slow hardwares. Specifically, for each issue, we carefully read the failure report to obtain an overview of the failure. If the unit test is provided, we run the unit test to reproduce the failure. The unit test provides simplified buggy logic, which is an important guideline for us to read source codes. If the unit test is absent, we directly read source codes based on conversations between the bug reporter and project maintainers. We can identify the root cause in both scenarios. If the root cause is related to the delay and the issue report explicitly mentions the delay is caused by the fail-slow hardware, we preserve this issue.

Table 2. FSH Failure Symptoms.

Symptom	%
Node service unavailable	58.3
Data loss	10.4
Data inconsistency	10.4
Job failure	8.3
Performance degradation	6.3
Client stuck	4.2
Node crash	2.1

Threats to Validity. Like all characteristic studies, the results of our study should be interpreted with the following limitations.

Representativeness of the selected systems. The selected systems are diverse, widely-used and open-source: ZooKeeper [76] is a distributed coordination service; MapReduce [64], HDFS [2], and HBase [27] are the cores of the dominant Hadoop data analytics platforms; Cassandra [4] is a highly available peer-to-peer NoSQL database. However, without accurate market information, it is difficult to conclude whether we have chosen the most widely-used cloud systems. Besides, closed-source cloud systems could have different characteristics.

Limitations of the filtering criteria. We clarify that this paper focuses on two types of fail-slow hardwares including the fail-slow storage and network devices. The main reason is that it is relatively easy for developers to confirm the impact of the fail-slow storage and network devices, i.e., slow I/O operations. It is still possible to miss the FSH failures whose issue reports do not contain the selected keywords. The fail-slow hardware is difficult to identify [25]. Developers may not be sure whether the delay is caused by the fail-slow hardware. Hence, the issue report possibly misses the selected keywords. We did try other keywords like "slow", and found that the resulting issue reports contain too many false positives, i.e., non-FSH failures. It is impractical to reduce false positives one by one.

Observer errors. To minimize the possibility of observer errors, each failure is investigated by two inspectors with the same criteria. Any disagreement is discussed in the end to reach a consensus.

3 Understanding FSH Failures

In this section, we present some findings and implications by analyzing collected FSH failures.

3.1 Findings

Finding 1: *Over half (58.3%) of FSH failures cause node service to be unavailable.*

Table 2 shows that FSH failures exhibit various failure symptoms. Over half of FSH failures cause certain software functionality or the entire node to be unavailability. In some cases, even a normal node is removed from the cluster. For example, in HDFS-9178 [32], the upstream datanode is stuck in the fail-slow disk. Hence, the downstream node timeouts when reading the packet from the upstream datanode. Then, the upstream datanode sends an ACK to the client and sets the downstream datanode status to `ERROR`. Finally, the client excludes the downstream datanode from the pipeline, even though the upstream datanode is the abnormal one.

Finding 2: *20.8% of FSH failures are silent (including Data loss and inconsistency).*

Table 3. FSH Failure Root Causes.

Root cause		**%**
Non-concurrency		81.2
◇ Indefinite blocking	41.6	
◇ Buggy internal checker	35.4	
◦ Buggy error checking	20.8	
◦ Buggy error handling	14.6	
◇ Infinite loop	2.1	
◇ Logic error	2.1	
Concurrency		18.8
◇ Data race	16.7	
◇ Deadlock	2.1	

These failures are difficult to detect without the correctness specification. For example, in HBase-26195 [28], synchronizing Hlog to HDFS timeouts due to the fail-slow hardware. The client receives an exception and rolls back the data protected by the Hlog. However, this procedure only rolls back the data in the primary node leading to data inconsistency between the primary and replicated nodes. Besides, internal checkers in HBase do not raise any explicit alarms in system logs or immediately stop the system, which makes this failure detection difficult.

Finding 3: *The root causes of studied failures are diverse. The top three (total 93.7%) root causes are indefinite blocking, buggy internal checker, and data race.*

Indefinite blocking means that synchronized tasks slowed down by fail-slow hardwares block other tasks for an uncertain time. In Figure 1, `writeRecord` within the `synchronized` block becomes extremely slow due to the fail-slow NIC. This synchronized `writeRecord` blocks all following write requests from clients. The fine-grained fail-slow NIC escapes the detection of `Heartbeat`, which is critical to trigger this failure. Buggy internal checker incorrectly handles timeout tasks caused by fail-slow hardwares. For example, in HDFS-5522 [31], the delay caused by the fail-slow disk is treated as a network error. Hence, `checkDiskError` is not invoked, which fails a lot of client requests. Data race occurs between timeout tasks caused by fail-slow hardwares and timeout handlers. Other root causes include infinite loop, logic error, and deadlock.

Finding 4: *Buggy internal checker fails to correctly handle timeout tasks.*

Buggy internal checker includes buggy error checking and error handling [73], which are confused by the impact of fine-grained FSH faults. Specifically, buggy error checking fails to figure out sources of FSH faults, such as fail-slow disks in HDFS-5522 [31]. Even though error checking is correct, error handling may also incorrectly handle FSH faults. For example, in MapReduce-1800 [52], the reducer fails to fetch the mapper's output due to the reducer-side fail-slow NIC. However, the reducer can still talk to the job tracker(JT). When a sufficient number of reducers fail, the normal mapper is blacklisted by the JT. In this example, although error checking finds the slow network, error handling incorrectly handles this fault. The fine-grained FSH fault is key to trigger this failure. Specifically, if a network partition occurs, the reducer cannot talk to the JT. Hence, the normal mapper is not blacklisted.

Finding 5: *Data race can occur between the timeout task caused by the fail-slow hardware and the corresponding timeout handler with simple timing constraints.*

Table 4. The percentage of each involved event.

Event type	%
Client/Server connection	12.5
Client request	83.7
One fail-slow hardware	89.6
Multiple fail-slow hardwares	10.4
Node reboot/join cluster	4.2

The task slowed down by the fail-slow hardware triggers the timeout mechanism. Moreover, the invoked timeout handler does not end the slow task. Therefore, if the timeout handler finishes before the slow task, this task will corrupt the data written by the timeout handler. For example, in ZooKeeper-710 [78], the client connects with the follower that encounters the fail-slow NIC. This connection request triggers the timeout mechanism. The invoked timeout handler sends another connection request to the leader. After successfully connecting with the leader, the former connection request is processed by the follower and the renewal request is sent to the leader. The renewal request overwrites the metadata of the current connection. As a result, the leader still connects with the client, but cannot process the client request. This data race occurs without the complex thread interleaving control. Besides, the fine granularity of the fail-slow hardware is necessary to trigger data race. Specifically, if coarse-grained faults, e.g., node crashes and network partitions, occur in the follower, the renewal request will not be sent to the leader. Hence, data race does not happen in this scenario. Additionally, our bug study shows that the buggy timeout mechanism is a new root cause for concurrency bugs.

Finding 6: *Most (89.6%) of FSH failures are caused by one fail-slow hardware.*

As shown in Table 4, the triggering conditions of most FSH failures do not require multiple fail-slow hardwares and complex inputs. For example, in HDFS-5341 [30], `DirectoryScanner` encounters a fail-slow disk when scanning the dataset with a lock. As a result, `DirectoryScanner` blocks the heartbeat and all DataXceiver threads that acquire the lock. This example only requires simple client requests as inputs.

3.2 Implications

Overall, our bug study reveals that the FSH failure is a severe problem in cloud systems (Table 2). Most (73.5%) of studied failures were reported in the last decade. Moreover, over half (58.3%) of FSH failures cause node service to be unavailable (Finding 1). Even though existing in-production detectors [33, 44, 48, 57] can detect part of FSH failures, the detected failures already cause damages, e.g., node unavailability and data inconsistency [33, 44, 48, 57]. Hence, it is necessary to design an FSH fault injection testing framework to detect FSH bugs before releasing production. The fault injection space explosion is a notorious problem in fault injection techniques [1, 6, 17, 19, 24, 36, 37, 47, 53]. However, our bug study reveals that synchronized and timeout-protected I/O operations are error-prone, which significantly reduces the fault injection space. Besides, most FSH failures are caused by one fail-slow hardware (Finding 6). Hence, the fault injection space does not increase exponentially due to the combinations of different fault points, which indicates that the fault injection testing framework can efficiently explore the fault injection space without complex fault injection strategies.

There are four points to avoid most FSH failures:

- Do not perform I/O operations within synchronization structures. For example, the solution of the example in Figure 1 is to copy and serialize data out of the `synchronized` block.
- Use the fine-grained lock to accurately synchronize I/O operations.
- Collect more information from other components of cloud systems to figure out sources of FSH faults, e.g., the delay from upstream datanode in HDFS-9178 [32], and take suitable actions.
- Avoid data race between the timeout task and timeout handler. Developers can use timestamps to avoid buggy execution order.

4 The Sieve Design

In this section, we first discuss Sieve's fault model (§ 4.1) and then describe the workflow (§ 4.2) and main components: Fault Point Analysis (§ 4.3), Injection Controller (§ 4.4), Workload Driver (§ 4.5), Failure Checkers (§ 4.6), and Agent-based Failure Validation (§ 4.7).

4.1 Fault Model

Fail-slow hardwares exhibit two types of fine-grained faults. The first type slows down partial I/O operations. The second type incurs exceptions from partial I/O operations. In this paper, we primarily focus on the first type. We additionally provide limited support for the second type via exception injection to simulate exception propagation caused by fail-slow hardware.

Cloud systems are designed to tolerate various faults, but their fault-tolerance levels are limited. If we inject multiple faults, the accumulated side effects possibly break cloud systems as expected. Moreover, one fault is enough to trigger most FSH failures (Finding 6). Hence, Sieve simulates the first type by injecting one delay for each test run.

4.2 Workflow

In a nutshell, Sieve identifies all fault points and then performs fault injection testing at each fault point. Figure 2 presents the workflow of Sieve, which consists of two phases. The first phase (left half of Figure 2) leverages the static analysis to identify fault points and instruments fail-slow agents within the cloud system to query fault injection decisions. The second phase (right half of Figure 2) treats the fault injection testing as a series of test runs. For each test run, Sieve leverages the workload driver to exercise the cloud system. When a fail-slow agent is executed, it sends an RPC query to the injection controller. The controller decides whether to inject the fault based on the history injection record. If the fail-slow agent receives a positive reply, it injects a delay into the cloud system. During this test run, failure checkers detect abnormal system behaviors, e.g., system crashes.

4.3 Fault Point Analysis

Cloud systems contain a large number of fault points affected by fail-slow hardwares. Moreover, most fault points can be tolerated by cloud systems [6, 19, 44, 47]. Therefore, it is impractical to exhaustively explore all fault points. Based on our bug study, we observe that synchronized and timeout-protected I/O operations are error-prone. Hence, our fault point analysis focuses on these two types of fault points, which can reduce the fault injection space.

The fault point analysis is outlined in Algorithm 1. Sieve constructs a new call graph, since the conventional one provided by Soot, a Java program analysis framework [65], is inefficient (Line 3). The call graph in Soot includes numerous irrelevant methods in Java Runtime Environment (JRE) and cloud systems. Based on the optimized call graph, Sieve identifies synchronized and timeout-protected I/O operations (Lines 4-18).

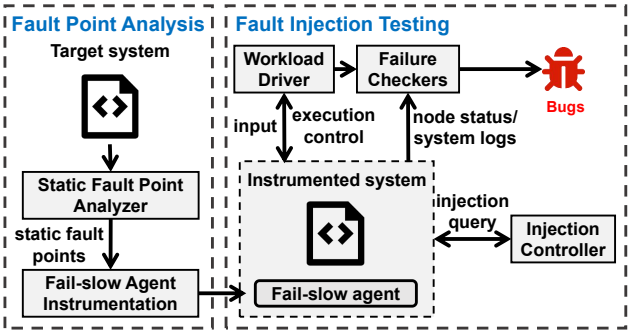

Fig. 2. The workflow of Sieve.

Identify the Synchronized I/O Operation. In general, synchronized mechanisms have two effects: (1) avoid multiple threads to simultaneously access critical data; (2) control different tasks to execute in a certain order. There are two corresponding patterns of synchronized mechanisms (Lines 23 & 27). The first pattern creates a critical region (Lines 22-25). For example, Lines 2-10 indicates a critical region in Figure 3. I/O operations (I/O_{1-3}) in the critical region are considered to be synchronized. To identify these synchronized I/O operations, Sieve needs to find the critical region. The critical region is wrapped by entry and exit points at the instruction level. It is non-trivial to identify the critical region by matching the entry point to the correct exit point, since branch and return instructions introduce more than one exit point. For example, in Figure 3, there are three exit points. If Sieve matches the entry point to the first or last exit points, it will miss a synchronized I/O operation (I/O_3) or incorrectly treat a non-synchronized I/O operation (I/O_4) to be synchronized.

To match the correct exit point, the intuitive method is to (1) understand how the programming language designs the interaction among the critical region, branch, and return instructions (2) and design a search algorithm that leverages former understanding. However, the first step is difficult and requires large human effort. Hence, we try another practical method. Specifically, we analyze different combinations of the critical region, branch, and return instructions case by case. In this process, we observe that the correct exit point is the last not followed by the return instruction, i.e., the second exit point in Figure 3. Based on this observation, Sieve can correctly identify the critical region and synchronized I/O operations in practice. Specifically, Sieve traverses the call graph twice. (1) If encountering an entry point, Sieve pushes this entry point into a stack. Subsequently, if encountering an exit point not followed by the return instruction, Sieve pops the top non-entry point from the stack and pushes this exit point into the stack. If the encountered exit point is followed by the return instruction, Sieve just ignores this exit point. Finally, all entry and matched exit points stored in the stack confirm all critical regions. (2) Sieve identifies I/O operations in critical regions.

The second pattern creates a hard barrier that makes sure one task is executed before the other (Line 26-29). For example, `notify()` and `wait()` create a barrier in Figure 4. I/O operations (I/O_1) before the barrier are considered to be synchronized. Identifying these synchronized I/O operations is trivial. Sieve traverses the call graph, finds barriers, and identifies I/O operations before these barriers. Timeout mechanisms also have this pattern that creates a soft barrier (using `notify()` and `wait(timeout value)`) that means the latter task ($Task_2$) can timeout and invoke timeout handler to process the former task ($Task_1$) (Line 39-42). Sieve uses the same method to deal with this pattern of timeout mechanisms.

Algorithm 1: Fault point analysis

```
1  Function identifyFaultPoint(sysClass, sootCallGraph):
2      ioOps ← parseIOOps(sysClass);
3      callGraph ← callGraphConstruction(ioOps, sootCallGraph);
4      syncScopes ← parseSyncScopes(callGraph);
5      timeoutScopes ← parseTimeOutScopes(callGraph);
6      syncColoredMethods ← colorMethodViaScopes(syncScopes, callGraph);
7      timeoutColoredMethods ← colorMethodViaScopes(timeoutScopes, callGraph);
8      candidateFaultPoints ← NULL;
9      foreach ioOp ∈ ioOps do
10         if syncColoredMethods.contains(ioOp) then
11             candidateFaultPoints.put(ioOp, SYNC);
12             continue;
13         end
14         if timeoutColoredMethods.contains(ioOp) then
15             candidateFaultPoints.put(ioOp, TIMEOUT);
16         end
17     end
18     return candidateFaultPoints;
19 Function parseSyncScopes(callGraph):
20     syncScopes ← NULL;
21     foreach method ∈ callGraph do
22         if isFirstSyncPattern(method) then
23             startLine, endMethod, endLine ← parseFirstSyncPattern(method, callGraph);
24             syncScopes.put({method, startLine}, {endMethod, endLine});
25         end
26         if isSecondSyncPattern(method) then
27             matchedMethod, startLine ← parseSecondSyncPattern(method, callGraph);
28             syncScopes.put({matchedMethod, startLine}, NULL);
29         end
30     end
31     return syncScopes;
32 Function parseTimeOutScopes(callGraph):
33     timeoutScopes ← NULL;
34     foreach method ∈ callGraph do
35         if isFirstTimeOutPattern(method) then
36             startLine, endMethod, endLine ← parseFirstTimeOutPattern(method, callGraph);
37             timeoutScopes.put({method, startLine},{endMethod, endLine});
38         end
39         if isSecondTimeOutPattern(method) then
40             matchedMethod, startLine ← parseSecondTimeOutPattern(method, callGraph);
41             timeoutScopes.put({matchedMethod, startLine}, NULL);
42         end
43     end
44     return timeoutScopes;
```

Identify the Timeout-protected I/O Operation. Timeout mechanisms are commonly used to handle unexpected faults in cloud systems. After investigating several cloud systems, we conclude two general patterns of timeout mechanisms (Lines 36 & 40). One has been discussed in § 4.3.

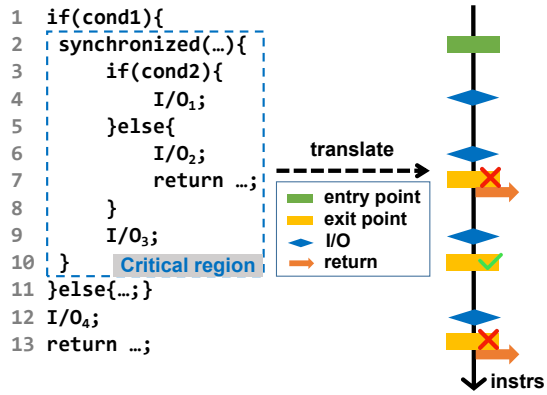

Fig. 3. The left part shows an example of creating a critical region (Lines 2-10). The right part shows low-level instructions, which only preserve I/O, return, entry and exit points.

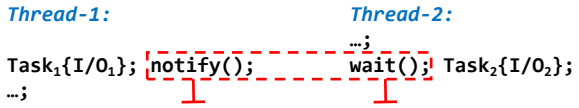

Fig. 4. An example of controlling the execution order of $Task_{1,2}$ via the barrier.

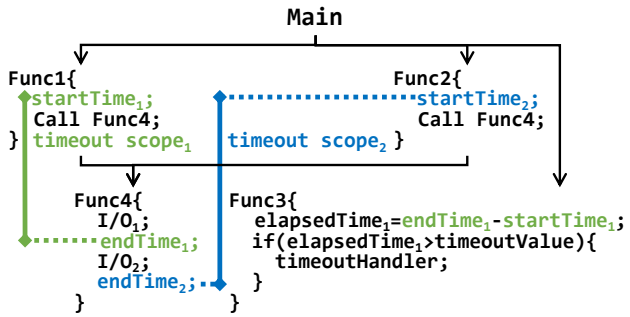

Fig. 5. An example of identifying timeout-protected I/O operations.

The other is related to get-time functions, e.g., `nanoTime` and `currentTimeMillis` in Java. Specifically, cloud systems record **startTime** and **endTime** assigned by get-time functions and calculate **elapsedTime = endTime - startTime** to obtain the execution time of the task. If elapsedTime exceeds the predefined timeout value, the timeout handler is invoked to process this abnormal task. We define the timeout scope as the scope between assignments of startTime and endTime (Lines 39-42). I/O operations in the timeout scope are considered to be timeout-protected. To identify these timeout-protected I/O operations, Sieve needs to match startTime to the corresponding endTime, which is non-trivial. In real-world cloud systems, there are many {**startTime**,**endTime**} pairs interleaving with each other, e.g., {$startTime_1$,$endTime_1$} interleaves with {$startTime_2$,$endTime_2$} in Figure 5. Moreover, most {startTime, endTime} pairs usually have irregular variable names. Hence, we cannot distinguish which endTime ($endTime_{1,2}$) is matched to

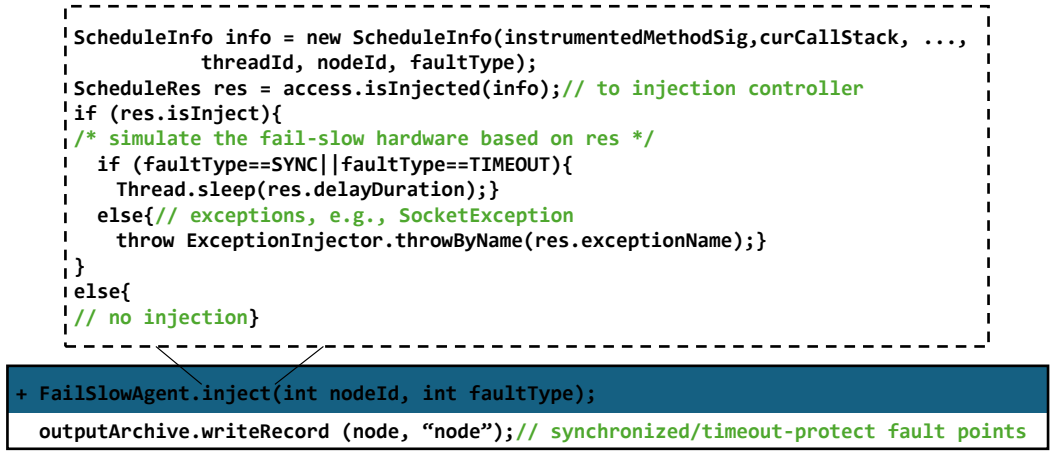

Fig. 6. The fail-slow agent instrumented for codes in Figure 1.

$startTime_1$. When reviewing this pattern, we note that startTime and the matched endTime must appear in the same formula: **elapsedTime = endTime - startTime**. Based on this observation, Sieve can correctly identify timeout-protected I/O operations. Specifically, Sieve (1) collects all variables ($startTime_{1,2}$ and $endTime_{1,2}$) assigned by get-time functions as a get-time set. (2) then checks all expressions whose forms are **A=B-C**. If two operands in the expression are in the get-time set and the result of this expression is used in `If` expression, this expression is the target formula ($elapsedTime_1 = endTime_1 - startTime_1$). The two operands ($startTime_1$ and $endTime_1$) in this expression correctly confirm a timeout scope ($timeoutscope_1$). (3) finally identifies all I/O operations in the timeout scope.

Instrument the Fail-Slow Agent. As shown in Figure 6, to enable fine-grained fault injection, Sieve instruments fail-slow agents before candidate fault points, i.e., synchronized and timeout-protected I/O operations. When the fail-slow agent is reached at runtime, it sends a query to the injection controller, which includes the current call stack, instrumented function information, instrumented position, thread ID, nodeID and fault type. If the reply is positive, the fail-slow agent simulates the FSH fault.

Sieve follows the principle of separation of mechanism and policy [26]. Specifically, the fail-slow agent serves as the mechanism, capable of independently simulating various faults, e.g., delay and data corruption. Meanwhile, the injection controller serves as the policy, which can independently develop different fault injection strategies to efficiently explore the fault injection space. For example, the injection controller determines the delay duration based on the fault type, such as synchronized and timeout-protected fault points. Thanks to this design, it is convenient to apply Sieve in complex failures. The fail-slow agent simulates degraded performance of fail-slow hardware by invoking *Thread.sleep* shown in Figure 6.

In addition to delay injection, Sieve supports exception injection. When an FSH fault affects an I/O operation, the fail-slow agent can inject exceptions that the corresponding I/O operation can throw in practice, thereby simulating exception propagation caused by fail-slow hardware. The injected exception type is determined based on the semantics of the affected I/O operation (e.g., SocketException and FileNotFoundException), allowing Sieve to more accurately reproduce real-world FSH fault behaviors. In our implementation, the fail-slow agent dynamically loads the

Algorithm 2: Fault injection strategy

```
1  Function scheduleFaultInjection(request):
2      if !checkRequest(request) then
3          return false;
4      end
5      fault ← getFault(request);
6      if fault.type==SYNC then delayTime ← 5 * 60 * 1000 ;
7      else delayTime ← 3 * getTimeOutValue(fault)/2 ;
8      grant(request,delayTime);
9  Function checkRequest(request):
10     if isInjected then
11         return false;
12     end
13     curContext ← getContext(request);
14     curFault ← getFault(request);
15     foreach preContext ∈ historyInfo.contexts() and preFault ∈ historyInfo.faults() do
16         if preFault=curFault and preContext.contain(curContext) then
17             return false;
18         end
19     end
20     record(historyInfo,curContext,curFault);
21     return true;
```

exception class by its name using Java reflection, instantiates the exception at runtime, and throws it at the fault point.

4.4 Injection Controller

In the fault injection phase, Sieve injects one fault for each test run. In each test run, the injection controller receives many injection requests from fail-slow agents, collects the information from injection requests, and decides on the fault injection. Finally, only one of the agents obtains a positive reply. To efficiently explore candidate fault points, Sieve implements grouping and context-sensitive injection strategies.

Grouping Injection Strategy. Through the fault point analysis, Sieve has pruned many fault points with a low possibility of triggering FSH bugs. However, there are still a large number of fault injection choices. It is inefficient to exhaustively explore all choices. We note many fault injections explore similar scenarios when these fault points are within the same basic block. Because different I/O operations within the same basic block are always under the same high-level system states (e.g., leader election and snapshot synchronization) and protected by the same fault-tolerance handler. To further reduce the fault injection space, Sieve groups fault points within the same basic block and selects the last fault point as the representative of the group. The fault point analyzer implements this grouping strategy by instrumenting the fail-slow agent before the last fault point within the same basic block.

Context-sensitive Injection Strategy. In addition to grouping fault points, Sieve implements a context-sensitive strategy (listed in Algorithm 2) to (1) find the buggy context of the fault point (2) and avoid repeatedly injecting faults at the same point under the same context. Specifically, Sieve collects the call stack of the fault point carried by the injection request (Line 20). When an injection request is received, Sieve checks whether the corresponding fault point has already been explored.

If the fault point is not explored, Sieve grants this fault injection. Otherwise, Sieve compares call stacks of previous fault injections with that of current fault injection (Lines 15-19). If the current call stack is unique, Sieve injects one fault at this point.

Fault Impact. Except for injection strategies, Sieve needs to determine how long the delay lasts. Specifically, Sieve checks the type of the fault point after receiving the injection request (Line 5). If the injection request comes from the synchronized I/O operation, Sieve uses 5 minutes as the default delay which is enough to cause indefinite blocking (Line 6). Hence, the gray failure checker in Sieve can detect this bug. Meanwhile, this delay duration does not significantly extend the evaluation time.

For the timeout-protected I/O operation, the delay duration is two-thirds of the timeout value (Line 7). In this way, Sieve not only triggers timeout mechanisms to mislead internal checkers but also reorders timeout tasks and timeout handlers. Sieve obtains timeout values by analyzing two general patterns shown in § 4.3 and instrumenting hooks. When hooks are reached at runtime, they send timeout values to the injection controller.

4.5 Workload Driver

Sieve leverages the workload driver to exercise a target cloud system. The workload driver can use various sources ranging from simple unit tests to carefully-crafted test cases as workloads. For each cloud system, we either implement several common workloads or select several existing test cases as workloads shown in Table 6.

The workflow of the workload driver is as follows. First, the workload driver creates multiple clients and each client connects with one system node. Hence, if the client encounters an exception, it can send an RPC request to failure checkers reporting the corresponding system node is suspicious. Second, the client executes the workload. If the client finishes the workload, it sends an RPC request to failure checkers reporting the workload progress. Finally, the workload driver restarts the target system for each test run to avoid the impact of previous injected faults.

4.6 Failure Checkers

To determine whether a bug occurs in cloud systems, Sieve provides three checkers:

- **Log error checker.** It scans the system log of each node to check whether there are FATAL, ERROR, and WARN entries or uncommon exceptions.
- **Gray failure checker.** It is a simplified version of Panorama [33] to identify whether differential observability exists. This checker marks the test run as suspicious if (1) the system's internal checker indicates a node is healthy, but the node's client reports errors. Specifically, this checker receives an RPC request that reports errors from the client, while obtaining a healthy status of the corresponding node by existing tools, e.g., `./zkServer.sh status` in ZooKeeper. (2) a subset of clients do not finish their workloads. Specifically, this checker does not receive RPC requests that report the workload progress from a subset of clients.
- **Data corruption checker.** Before each test run, the checker persists the relevant data, e.g., znode in ZooKeeper, as $Data_B$. Sieve executes the workload without injecting faults, and this checker saves the current data as $Data_A$. $Data_B$ and $Data_A$ collectively serve as the reference for correct system behaviors. This checker only records $Data_B$ and $Data_A$ once, since the workload remains consistent across test runs. Therefore, the overhead of this checker is negligible. After each test run with the fault injection, this checker saves the resulting data as $Data_T$ and compares it with $Data_B$ and $Data_A$. If any elements in $Data_T$ are different from those in $Data_B$ and $Data_A$, this checker marks the corresponding test run as suspicious.

Table 5. Skill specification for agent-based failure validation

Field	Description
Name	FSH Failure Validation
Description	Determine whether a suspicious test run reported by Sieve corresponds to a real FSH failure.
FSH Failure	FSH failure definition and symptoms
Validation Workflow	(1) Analyze checker reports; (2) Inspect system logs; (3) Cross-check observed behaviors against the design documentation of the system under test.
Criteria	A test run is classified as a real FSH failure if the observed behavior violates expected system semantics and cannot be explained by fault-tolerance mechanisms.
Input	Checker reports, system logs, and relevant design documentation.
Output	A structured verdict: `TRUE_BUG`, `FALSE_POSITIVE`, or `UNCERTAIN`, together with an explanation.

These checkers generate a simplified report that indicates which predefined rules are violated. Moreover, as a testing framework, developers can flexibly add specific checkers to assert system properties they care about, such as performance bug [40] and inconsistency [45].

4.7 Agent-Based Failure Validation

To reduce the substantial manual effort for failure validation, Sieve provides an optional agent-based failure validation component. For each suspicious test run reported by failure checkers, the agent determines whether the system under test indeed encounters an FSH failure by jointly analyzing checker reports, system logs, and the design documentation of the system under test.

Rather than building a new agent framework, Sieve directly leverages existing agent systems, such as opencode, Codex, and Claude Code [9, 10, 54]. We provide a dedicated skill that guides the agent in deciding whether a real FSH failure occurs. This skill explicitly specifies the knowledge and reasoning process required for validation, including seven components shown in Table 5. Specifically, after learning the characteristics of FSH failures, the agent follows a three-step validation process: (1) it analyzes checker reports; (2) it inspects system logs, (3) it cross-checks observed behaviors against the design documentation of the system under test. Based on predefined criteria, the agent produces a structured validation result. According to the agent's output, developers can significantly narrow down their manual validation effort by focusing on test runs classified as `TRUE_BUG` or `UNCERTAIN`, while deprioritizing those classified as `FALSE_POSITIVE`. This human-in-the-loop workflow allows the agent to systematically reduce broad manual inspection effort without eliminating human judgment.

5 Implementation

We have implemented Sieve in Java with around 8,100 SLOC for core components. The fault point analyzer is built on top of Soot, a Java program analysis framework [65]. Sieve instruments cloud systems using Javassist, a Java bytecode instrumentation toolkit [35]. The injection controller is designed in a client-server architecture via Java RMI for RPCs.

I/O Operation Identification. Sieve identifies all I/O operations by statically analyzing cloud systems. If the function is implemented by general I/O packages, it is treated as an I/O operation.

Sieve analyzes five general I/O packages, including `java.io`, `java.nio`, `java.net`, `javax.net` and `io.netty`. Additionally, some cloud systems implement their customized I/O operations, e.g., serialize/deserialize APIs in class Record of ZooKeeper. Sieve also identifies such I/O operations.

Algorithm 3: Call Graph Construction

```
1  Function callGraphConstruction(ioOps, sootCallGraph):
2      callGraph ← NULL;
3      foreach ioOp ∈ ioOps do
4          recur(ioOp, callGraph, sootCallGraph);
5      end
6      return callGraph;
7  Function recur(ioOp, callGraph, sootCallGraph):
8      if !isAnalyzed(ioOp, callGraph) then
9          callSites ← sootCallGraph.edgesInto(ioOp);
10         foreach caller ∈ callSites do
11             if isSystemMethod(caller) then
12                 lineNum ← getLineNum(caller, ioOp, sootCallGraph);
13                 callersMap.put(caller, lineNum);
14                 if callGraph.forward.get(caller)==NULL then  callGraph.forward.put(caller, ioOp) ;
15                 else  callGraph.forward.get(caller).add(ioOps) ;
16             end
17         end
18         callGraph.backward.put(ioOp, callersMap);
19         foreach caller ∈ callSites do
20             recur(caller, callGraph, sootCallGraph);
21         end
22     end
```

Call Graph Construction. Soot provides a call graph that includes substantial irrelevant information (e.g., constructors like *init()*), making it inefficient for identifying synchronized and timeout-protected fault points. To address this, Sieve constructs an efficient call graph that only contains I/O operations and their callers. The core idea is that Sieve constructs the call graph from I/O operations and omits irrelevant methods from Java Runtime Environment (JRE) and cloud systems, which is listed in Algorithm 3. Specifically, Sieve begins by identifying I/O operations. To avoid infinite loops, Sieve identifies whether each I/O operation has already been processed (Line 8). If the I/O operation has been analyzed, Sieve skips it. Otherwise, the cycle in the call graph of Soot causes Sieve to fall into infinite loops. For the new I/O operation, Sieve retrieves its callers but filters out those belonging to JRE (Line 11). The remaining callers and invocation positions of the I/O operation are saved in *callersMap* (Lines 12-13). *callersMap* and the corresponding I/O operation are saved in the new call graph, i.e., *callGraph* (Line 18). To construct the complete call graph, Sieve recursively obtains all callers of the I/O operation (Line 20). It is worth noting that Sieve does not obtain callees of the I/O operation. There are two reasons. (1) The callees are possibly irrelevant to I/O operations. If so, the callees cause the performance overhead when identifying synchronized and timeout-protected fault points. (2) If the callees are related to I/O operations, they must be callers of some I/O operations included by *ioOps* (Line 1). Therefore, Sieve does not need to analyze callees of the I/O operation. Additionally, the optimized call graph supports bidirectional querying that benefits fault point identification (Lines 14-15).

Algorithm 4: Coloring

```
1  Function colorMethodViaScopes(scopes, callGraph):
2      curColorList ← NULL;
3      coloredMethods ← NULL;
4      foreach startPoint, endPoint ∈ scopes do
5          curColorList.append(methodsWithinScope(startPoint, endPoint, callGraph));
6      end
7      foreach colorMethod ∈ curColorList do
8          callees ← getAllCallees(colorMethod, callGraph);
9          coloredMethods.append(callees);
10     end
11     return coloredMethods;
12 Function getAllCallees(method, callGraph):
13     tmpCallees ← callGraph.forward.get(method);
14     callees.append(tmpCallees);
15     foreach callee ∈ tmpCallees do
16         tmpRes ← getAllCallees(callee, callGraph);
17         callees.append(tmpRes);
18     end
19     return callees;
```

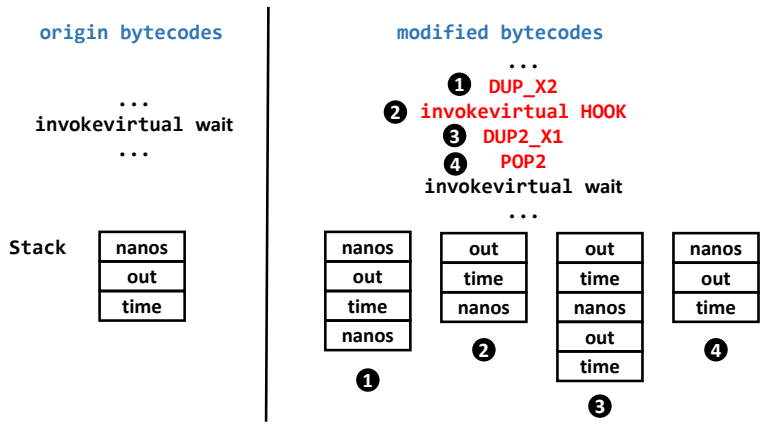

Fig. 7. An example of collecting timeout values in wait (long timeout, int nanos).

Fault Point Identification. In Section § 4.3, we present the details of identifying critical regions and timeout scopes (CRTS) in synchronized and timeout mechanisms. Sieve proceeds to identify synchronized and timeout-protected I/O operations, as listed in Algorithm 4. Specifically, Sieve obtains all methods located within the identified CRTS and their callees through the call graph. Identifying methods in CRTS needs to respect the precise boundaries specified by the *startLine* and *endLine* stored in the call graph. Otherwise, irrelevant I/O operations, e.g., the node failover I/O, are included in the testing scope. Finally, Sieve identifies synchronized and timeout-protected I/O operations from the methods and their callees.

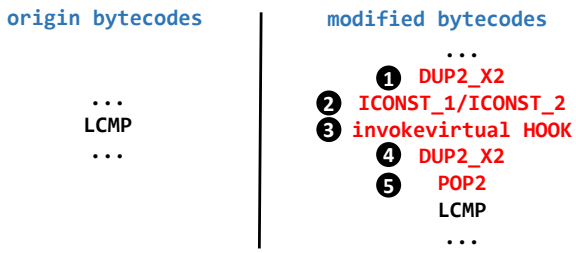

Fig. 8. An example of collecting timeout values in LCMP instruction.

Timeout Value Collection. To collect timeout values, Sieve modifies Java bytecodes of cloud systems using Javassist. There are three steps. (1) Sieve identifies Java bytecodes that confirm timeout scopes, based on timeout scope analysis in Section § 4.3. (2) Sieve instruments hooks into Java bytecodes to obtain timeout values. The hooks must guarantee the correctness of the stack frame. (3) Sieve sends timeout values to the injection controller.

Specifically, Sieve analyzes two general patterns of timeout mechanisms presented in Section § 4.3. For the first pattern, Sieve instruments hooks before soft barriers, e.g., `wait` in Java. The hooks intercept barriers and capture their parameters. As shown in the left part of Figure 7, the parameters of *wait (long timeout, int nanos)* are stored on the stack. According to the convention of Java Virtual Machine (JVM), the parameters are pushed onto the stack from left to right. The right part of Figure 7 shows four injected bytecodes and their corresponding stack states. *DUP_X2*, *DUP2_X1*, and *POP2* are used to guarantee the correctness of the stack frame during collecting timeout values. Specifically, Sieve utilizes *DUP_X2* to duplicate *nanos* and insert it below *timeout* (❶). The hook consumes both *nanos* and *timeout* from the stack and leaves *timeout* as the returned value on the stack (❷). *DUP2_X1* duplicates *timeout* and places it below *nanos* (❸). *POP2* pops the top *timeout* from the stack (❹). In the end, the stack is restored to its initial state. Sieve collects timeout values from the second pattern using a similar method shown in Figure 8. Sieve finds elapsedTime through the taint analysis and instruments a hook before `CMP` instruction that contains elapsedTime. The hook captures the other operand of `CMP` instruction. It is worth noting that both operands of CMP instruction may represent timeout values. Therefore, *ICONST_1* and *ICONST_2* are used to indicate the operand corresponding to the timeout value. The captured parameters and operands are timeout values. Finally, the hooks send timeout values to the injection controller.

Memory Access Tracing. Sieve instruments all methods at the bytecode level to trace heap-based shared memory accesses. The tracer intercepts field read and write operations (i.e., getfield, putfield, getstatic, and putstatic) in the Java bytecode. For each traced access, Sieve records the accessed field, source-code line number, executing method, thread identifier, and node identifier. To preserve correctness of the original execution, Sieve carefully inserts stack-manipulation bytecodes (e.g., *DUP*, *DUP2_X1*) to keep the operand stack consistent after instrumentation. The collected traces can be used to identify potential conflicting read–write or write–write accesses, helping developers diagnose concurrency-related FSH bugs.

6 Evaluation

Our evaluation aims to answer two questions: (1) How effective is Sieve in detecting bugs? (3) How does Sieve compare with state-of-the-art fault injection approaches?

Table 6. The evaluated systems.

System	Release	Workload
ZooKeeper (ZK)	3.9.0	Create/read/update/delete znode
Kafka (KA)	3.6.0	Producer/consumer performance test
HDFS	3.3.6	Read/write/move/put file

Table 7. Bugs detected by Sieve. The last five columns show whether the bugs are detected by different approaches shown in Table 9.

	Bug ID	Failure Symptoms	Random	FATE	Legolas	Chronos	Sieve-I	Sieve-S	Sieve
Unknown Bugs	ZK-4816	A follower cannot follow the leader for a long time (more than 30 seconds)	✗	✗	✗	✗	✗	✓	✓
	ZK-4817	CancelledKeyException cannot catch the client disconnection exception	✗	✗	✗	✓	✗	✓	✓
	ZK-4844	Fail-slow disk while executing writeLongToFile causes the follower to hang	✗	✗	✗	✗	✗	✓	✓
	ZK-4836	Inconsistent ACL index leads to MarshallingError	✗	✗	✗	✗	✗	✓	✓
	KAFKA-16401	One request consumes all request handler threads	✗	✓	✓	✗	✓	✓	✓
	KAFKA-16412	An uncreated topic is considered as a created one	✗	✗	✓	✓	✗	✓	✓
Known Bugs	HDFS-15869	Namenode hangs due to the slow sendResponse	✗	✓	✓	✗	✓	✓	✓

Evaluated Systems. We evaluated Sieve by using three widely used and open-source cloud systems in Table 6. Note that Kafka [38] is not included in our bug study, which is a popular stream-processing service. To prove that Sieve is not specific to the five studied systems, we selected Kafka as a benchmark. All target systems are the latest versions during the experiments.

Setup. We apply default configurations to all cloud systems and deploy a cluster of cloud systems on a single physical machine using dockers. The physical machine contains two Intel Xeon Gold 6230R CPUs, which include 52 cores, and 192GB DRAM running Ubuntu18.04. In the evaluation, the cluster consists of either three nodes for ZooKeeper and Kafka or four nodes for HDFS. The fault injection experiment for each system consists of 2000 test runs. The time of each test run varies depending on how the system reacts to the injected faults and whether it fails early or not. The experiment time for the three systems is 20.9 hrs, 24.9 hrs, and 29 hrs respectively.

6.1 Effectiveness of FSH Bug Detection

6.1.1 Methodology. We apply Sieve to the target systems and check whether Sieve can detect unknown bugs and reproduce studied bugs. Additionally, we evaluate the contribution of different failure checkers in Sieve by checking how many bugs each checker can detect.

6.1.2 Detecting Unknown Bugs. As shown in Table 7, Sieve has detected seven FSH bugs, including six unknown bugs and one known bug (HDFS-15869). Moreover, ZK-4836 and KAFKA-16412 have been confirmed by developers. Although HDFS-15869 was reported by developers, it has not been fixed yet. Hence, Sieve can detect HDFS-15869. Moreover, we did not know about HDFS-15869 before. These bugs cause node hang, uncaught exceptions, node crash, and semantic violation.

ZK-4816. A follower cannot join the cluster for a long time (more than 30 seconds). Specifically, the cluster consists of three nodes. The first node becomes the leader after the election. The leader is stuck in serializing the snapshot to the local fail-slow disk. The followers disconnect from the leader, since their sync operations are blocked by the snapshot serialization in the leader. These two followers start a new leader election. The second node becomes the new leader. However, the old leader does not change the node status. There have been two leaders in the cluster for a long time. The third node is confused and cannot follow any leader for a long time.

ZK-4817. `CancelledKeyException` cannot catch the client disconnection exception in some cases. Specifically, NIOServerCxn throws `CancelledKeyException` if the client disconnects from

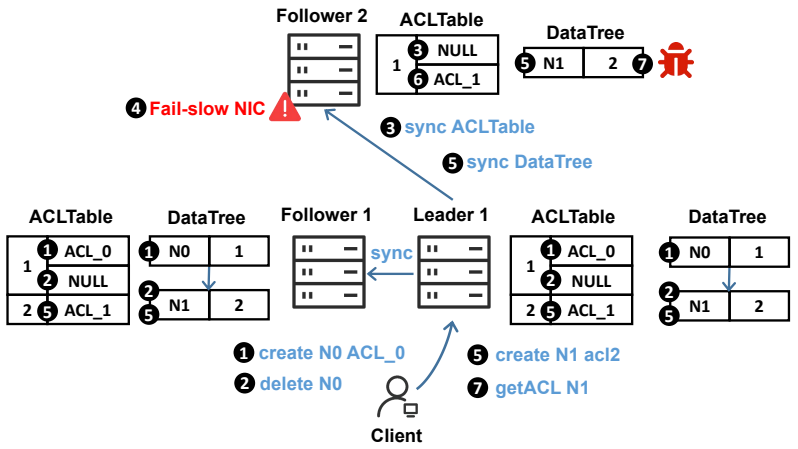

Fig. 9. The details of ZK-4836.

the server abnormally. When NIOServerCxn executes doIO, the disk becomes fail-slow. The client cannot receive heartbeats and disconnects from the server. If doIO is blocked for 30 seconds, NIOServerCxn throws CancelledKeyException. However, when doIO is stuck for more than 30 seconds, CancelledKeyException disappears in system logs. Although the clients disconnect from the server abnormally in two scenarios, CancelledKeyException does not work in the latter scenario.

ZK-4844. Fail-slow disk while executing writeLongToFile causes the follower to hang. Specifically, the follower executes writeLongToFile slowed down by the fail-slow disk. The internal checker is blocked by writeLongToFile. The leader excludes the follower from the cluster, but the follower believes that it still acts as a follower. As a result, all requests sent to the follower cannot be processed. This bug is similar to Zookeeper-4074 [77]. Zookeeper-4074 can be solved by adding -Dlearner.asyncSending=true. However, this method cannot solve ZK-4844.

ZK-4836. An inconsistent ACL index leads to MarshallingError shown in Figure 9. Specifically, when a leader creates a node (N0) with an ACL entry (ACL_0) and deletes it due to the node deletion (assume the current aclindex is 1) (❶ and ❷). A follower (Follower 2) joins into the cluster and starts to synchronize the snapshot (including the ACL table and datatree) with the leader. When synchronizing the ACL table to the follower (❸), the NIC becomes fail-slow (❹), which slows down the ACL table transmission. Meanwhile, a client sends a request that creates a node (N1) with a new ACL entry (ACL_1) to the leader (❺). In the leader, the datatree contains N1->2 (indicates that N1 points to the second slot of the ACL table) and the ACL table contains 2->ACL_1 (indicates that ACL_1 is stored in the second slot of the ACL table). The leader synchronizes the updated datatree to the follower. When the follower deserializes the new datatree, the aclindex is set to the number of elements in the ACL table. The follower contains the old ACL table (aclindex=1) and the new datatree (N1->2). The ZAB protocol synchronizes the client creation request from the leader to the follower. When replaying the client creation request in the follower, ACL_1 will be added into the first slot of the ACL table (1->ACL_1) (❻). However, the node N1 still points to the second slot of the ACL table (N1->2). Finally, when executing getAcl N1 (❼), MarshallingError arises, which leads to the follower crash.

KAFKA-16401. One request consumes all request handler threads. Specifically, a consumer request is stuck in storeOffsets slowed down by the fail-slow disk. The thread response for this

Table 8. Bug detection breakdown by failure checkers.

	Log error checker	Gray failure checker	Data corruption checker
Unknown Bugs	2	4	0
Studied Bugs	6	25	5

request is blocked and holds a group lock. Due to timeouts, the consumer resends requests to the broker. All request handler threads are quickly occupied by these requests and stuck in acquiring the group lock. As a result, new requests cannot be processed.

KAFKA-16412. An uncreated topic is considered as a created one. Specifically, a client sends the request for topic creation to the broker. Another client also sends the same request to the broker. When the first request is not finished, the second one fails and returns `TopicExistsException` to the second client. However, subsequent requests that operate on this unestablished topic from the second client fail, which confuses the second client.

HDFS-15869. Namenode hangs due to the slow `sendResponse`. Specifically, a client sends a write request. When the namenode handles this request and writes the editlog to the disk, the disk becomes fail-slow. The thread response for writing the editlog is stuck, so that it does not process the following write requests from clients.

Except for five detected bugs in ZooKeeper and HDFS, Sieve also detects two bugs in Kafka that are out of the five studied systems. Because (1) Sieve is designed to detect FSH bugs for most cloud systems that contain synchronized and timeout mechanisms, not just the studied systems. (2) There are buggy synchronized and timeout mechanisms in Kafka.

6.1.3 Reproducing Studied Bugs. To further evaluate Sieve's effectiveness, we try to reproduce all studied bugs. For each bug, we first check whether Sieve can correctly identify its corresponding fault point. If successful, we further modify Sieve and only inject the FSH fault at the corresponding point to detect the bug. If the failure report provides workloads, Sieve directly uses them. Otherwise, the workloads shown in Table 6 are used. When the checkers mark test runs suspicious, we manually check whether their symptoms and system logs are consistent with the original failure report in JIRA. If consistent, the corresponding bug is successfully reproduced.

Sieve successfully reproduces 36 of 48 studied bugs. There are 12 bugs not reproduced. For the five bugs (HBase-11536, HDFS-3493, HDFS-11755, HDFS-16659, and MapReduce-1800), multiple fault injections within one test run are necessary. However, Sieve currently focuses on a single fault injection. To reproduce these five bugs, we need to provide multiple buggy fault points and the correct fault injection order. Moreover, for the first four bugs, we need to provide additional checkers due to their silent symptoms (Finding 2). For four silent bugs (ZK-417, HDFS-10301, HBase-26195, and HBase-13430), Sieve fails to reproduce them due to the absence of accurate checkers. Additionally, to reproduce HDFS-10301, the fine-grained thread interleaving control is necessary. Although Sieve incorporates memory access tracing to identify potential conflicting memory accesses, selecting the correct conflicting access pair and enforcing the precise thread scheduling required to trigger the bug remain highly challenging.

For the bug (ZK-4293), the complex thread interleaving causes deadlock. However, Sieve is designed to detect concurrency bugs with simple timing constraints (Finding 5). Although Sieve fails to reproduce the above ten bugs, it can identify buggy fault points. However, Sieve fails to reproduce the remaining bugs (HBase-12270 and HBase-27947), since the static analysis in Sieve cannot analyze the synchronized data structures, i.e., the FIFO queue in these two bugs. To reproduce these two bugs, we need to extend the static analysis in Sieve to analyze the FIFO queue.

6.1.4 Effectiveness of Failure Checkers. We analyze how different failure checkers contribute to bug detection, in terms of the number of unknown and studied bugs they detect. Our results show that the three checkers are complementary.

Table 8 breaks down the detected bugs by the three failure checkers in Sieve. The checkers are executed in a fixed order: the gray failure checker runs first, followed by the data corruption checker, and finally the log error checker. This ordering reflects their roles and costs: the gray failure checker is an *online* checker that monitors runtime progress and client-observed symptoms during a test run, whereas the data corruption checker and the log error checker are *offline* checkers that analyze post-run artifacts (e.g., persisted states and logs). Moreover, the data corruption checker serves as a strict oracle—it compares post-run data states against reference states and introduces no false positives.

For *unknown bugs*, the gray failure checker detects three bugs (ZK-4836,ZK-4844, and KAFKA-16401) by observing that a subset of clients fails to complete the workload (i.e., missing expected RPC progress reports), and it detects ZK-4816 by identifying an abnormal cluster status with simultaneously active leaders. The log error checker detects ZK-4817 and KAFKA-16412 by identifying abnormal exceptions (CancelledKeyException and TopicExistsException) in system logs.

For *studied bugs*, the gray failure checker detects the majority of reproduced bugs (25/36), which is consistent with our bug study showing that node service unavailability dominates the failure symptoms (Finding 1). The data corruption checker detects 5 studied bugs. Among them, three also manifest log errors and are therefore detectable by the log error checker. However, the data corruption checker detects these three bugs without introducing false positives, significantly reducing the manual effort for failure validation. Moreover, the data corruption checker detects two silent bugs (HDFS-7065 and HDFS-13111) that do not manifest log errors or gray failure symptoms. The log error checker detects 6 studied bugs that manifest error entries or abnormal exceptions in system logs. Overall, these results demonstrate that combining multiple checkers is necessary to effectively detect diverse FSH failures. Online symptom-based checking is crucial for gray failures, while offline data and log analysis provide complementary coverage for data corruption and log errors.

6.2 Comparison with Alternative Fault Injection Approaches

6.2.1 Methodology. We compare Sieve with six alternative fault injection approaches shown in Table 9. These approaches include coarse-grained (FATE) and fine-grained (Random, Legolas, Chronos, and two variants of Sieve) tools.

Random is implemented as the baseline of the fault injection strategy. We reuse the client-server architecture of Sieve and instrument the fail-slow agent before each I/O operation. When the fail-slow agent is reached at runtime, the injection controller randomly grants an injection request.

FATE [24] focuses on node-level faults, making it not directly comparable to Sieve. We re-implement its fault injection strategy in Sieve to attempt meaningful comparisons. Specifically, we define the failure ID like FATE, which comprises the class name, function name, line number, and call stack of the I/O operation. The injection controller can receive injection requests from all I/O operations and grant an injection request whose failure ID is unique.

Legolas [68] is a state-of-the-art fine-grained fault injection tool. Legolas infers abstract states of cloud systems and stores fault points in queues. The fault injection strategy of Legolas is called budgeted-state-round-robin (bsrr). Specifically, Legolas selects a queue in the round-robin way and dequeues a fault point to test.

Chronos [8] is a state-of-the-art fine-grained delay injection tool and adopts the deep-priority guided algorithm to detect timeout bugs. However, its core components are not available. Hence, we re-implement its fault injection strategy in Sieve. Specifically, the injection controller records

Table 9. Settings for Alternative Approaches. S/T I/O operations represent synchronized and timeout-protected I/O operations.

Approach	Fault Point	Injection Strategy
Random	All I/O operations	Random
FATE [24]	All I/O operations	Context-sensitive
Legolas [68]	All I/O operations	Abstract state and bsrr
Chronos [8]	T I/O operations	Deep-priority
Sieve-I	S/T I/O operations	Context-insensitive
Sieve-S	S/T I/O operations	Context-sensitive
Sieve	S/T I/O operations	Grouping and context-sensitive

Table 10. # Dynamic/Static fault points indicate the number of injected and candidate faults. S/T indicates that Sieve identifies all synchronized and timeout-protected fault points. S/T+Gr indicates that Sieve applies the grouping strategy to S/T fault points.

	# Dynamic fault points							# Static fault points		
	Random	FATE	Legolas	Chronos	Sieve-I	Sieve-S	Sieve	All	S/T	S/T+Gr
ZooKeeper	2000	2000	2000	2000	423	1291	705	1905	1266	856
Kafka	2000	2000	2000	2000	267	1968	967	1953	1090	780
HDFS	2000	2000	2000	2000	383	883	658	4216	1974	1568

timeout-protected I/O points, randomly selects six fault points to explore and records new fault points that only appear after injecting delays. If the depths of new fault points are larger than the average depth of selected fault points, Sieve injects delays at new fault points and selected fault points in the next test run. The depth of a fault point means the number of preceding fault points that include tested and untested fault points in the execution path. If there is not a new fault point, the injection controller randomly selects six fault points again.

Sieve-I adopts the context-insensitive injection strategy which means that all identified fault points are explored once. Except for the injection strategy, other components of Sieve-I are the same as Sieve.

Sieve-S is built on top of Sieve-I. Sieve-S explores identified fault points multiple times. We evaluate the effectiveness of grouping and context-sensitive strategies by comparing Sieve-S with Sieve and Sieve-I respectively.

All the above six approaches are performed with the same settings as Sieve, i.e., cluster configurations, failure checkers, workload driver, and 2000 test runs that consume more than 20 hours. Additionally, Chronos injects multiple faults since its strategy is designed to explore the combinations of different delays. Except for Chronos, the remaining five approaches inject one fault for each test run. Legolas uses its fault point configuration, which injects a one-minute delay for each test run. Random and FATE also inject a one-minute delay for each test run like Legolas.

6.2.2 Comparison Result Analysis. Compared with alternative approaches, we aim to answer one question: is Sieve more effective and efficient in detecting FSH bugs? We answer this question from two aspects: the number of detected bugs and explored fault points.

The Number of Detected Bugs. Table 7 shows that Sieve is more effective in detecting FSH bugs. Random does not detect any bugs since it misses buggy fault points in deeper system states due to injecting faults at the beginning of workloads.

Compared with FATE and Legolas, four out of seven FSH bugs can only be detected by Sieve. FATE exhaustively explores the huge fault injection space, which is impractical and inefficient. Hence, FATE misses the buggy points of ZK-4816, ZK-4844, and KAFKA-16401 within 2000 test runs. Although we can add more test runs, FATE will generate more reports which require large human effort to confirm true bugs. Legolas fails to detect ZK-4816 and ZK-4844 due to the same reason. For ZK-4817, to observe different behaviors in system logs, injecting fault twice with different delay durations at the buggy fault point is necessary for the checkers. Although FATE and Legolas may explore this point multiple times, the duration of the injected delay does not change. Hence, FATE and Legolas fail to detect ZK-4817. ZK-4836 is a concurrency bug and requires a suitable timing constraint. However, the long delay (1min) injected by FATE and Legolas causes the client creation request to timeout, so that the updated datatree will not be synchronized to the follower. As a result, ZK-4836 does not occur. Therefore, FATE and Legolas fail to detect ZK-4836.

Compared with Chronos, Sieve detects five more bugs. ZK-4816, ZK-4844, and HDFS-15869 are out of the scope of Chronos since these bugs are not related to timeout mechanisms. Chronos fails to detect ZK-4836 since multiple delay injections cause the client creation request to timeout, like a single long delay injection in FATE and Legolas. For KAFKA-16401, Sieve injects a delay before the synchronized I/O point on the server side, which causes the client to frequently retry the timeout request. If Chronos wants to detect KAFKA-16401, it needs to repeatedly inject a delay before the same I/O point on the client side. However, this scenario contradicts the deep-priority guided algorithm of Chronos, which aims to explore the combinations of different delays. Chronos can detect ZK-4817 and KAFKA-16412 thanks to our broad log error checker. The original failure checkers of Chronos described in its paper detect two symptoms: server crash and service hang. However, ZK-4817 and KAFKA-16412 do not cause these two symptoms. Therefore, Chronos should have failed to detect the two bugs.

The Number of Explored Fault Points. We record the number of dynamic and static fault points in Table 10. The number of dynamic fault points is less than 2000, which indicates all fault points exposed under current workloads are explored. Tables 7 and 10 show that Sieve can explore fewer fault points, while detecting more bugs. However, Random, FATE, and Legolas explore too many useless fault points to detect some FSH bugs. In some worse cases, Random even repeatedly explores the same fault under the same runtime context. Compared with FATE, Sieve-S can efficiently detect more FSH bugs, since our bug study reveals that synchronized and timeout-protected I/O operations are error-prone, which significantly prunes the fault injection space. Besides, thanks to the effective context-sensitive strategy, Sieve-S explores more useful contexts of fault points and further detects more FSH bugs than Sieve-I. Moreover, compared with Sieve-S, Sieve explores fewer fault points without compromising the number of detected bugs, which indicates that the grouping strategy is effective. In summary, Sieve is more efficient in detecting FSH bugs thanks to its effective design.

Chronos aims to explore the combinations of different delays, which cause the fault injection space to increase in an exponential manner. Therefore, Chronos implements a deep-priority guided algorithm to efficiently explore such huge fault injection space. However, Sieve focuses on the practical and common scenario, i.e., one delay injection (Finding 6). Chronos and Sieve have different target scenarios. Hence, we do not compare with Chronos in this aspect.

6.3 False Positive

Threats of false positives come from two sources. The first source is injected faults. If the injected fault is not realistic, a false positive arises. Specifically, injecting delays before non-blocking I/O operations causes false positives. To eliminate these false positives, Sieve simulates a 10s delay for the fail-slow disk and NIC using the device mapper (DM) and traffic control (TC) in Linux (a

Table 11. Results of agent-based failure validation. NCC denotes the number of correctly classified test runs.

System	NCC	Average Validation Time (seconds)	Average Cost ($)
ZooKeeper	9	120.74	0.50

10s delay is suitable since it exceeds the time consumed by normal I/O operations and does not break cloud systems). As a result, the consumed time of the non-blocking I/O operation is less than 10 seconds. Because the non-blocking mechanisms are implemented above the layer that simulates the fault using DM and TC in Linux, which indicates the non-blocking I/O operation returns before reaching the simulated fault. Based on this feature, Sieve re-executes the suspicious test run marked by failure checkers and calculates the time consumed by the explored I/O operation. If the consumed time is less than 10 seconds, Sieve abandons the report of the suspicious test run. Overall, the faults injected by Sieve do not cause any false positives. In addition, the fault simulated by DM and TC slows down all I/O operations, which cannot enable fine-grained fault injection.

The second source is failure checkers. The gray failure checker is a simplified version of Panorama [33], so it cannot obtain comprehensive information of components in cloud systems. For example, in Zookeeper, if Sieve injects a delay on the client side when the client connects to the server, the gray failure checker will obtain an error report from the client. Meanwhile, the gray failure checker obtains a healthy status of the server by using `./zkServer.sh status` provided by Zookeeper. Based on different server statuses observed by the client and server, the gray failure checker reports a bug. In fact, the reported bug is a false positive. The log error checker also makes mistakes. For example, our checkers mark a test run suspicious when a fault is injected in `LearnerHandler` since ERROR entries appear in system logs. However, such ERROR entries are expected behaviors in ZooKeeper. To eliminate these false positives, for each suspicious test run, we (1) read system logs to understand system behaviors; (2) check whether system behaviors are consistent with design documentation. (3) report this bug to developers and discuss it with them. In the near future, we plan to enhance our checkers by using more accurate rules.

To reduce the manual effort for failure validation, we introduce an optional agent-based failure validation component. To evaluate its effectiveness, we collect 10 suspicious test runs from ZooKeeper, including six false positives and four unknown bugs reported in Table 7, and validate them using an agent implemented with opencode and claude-opus-4.7, guided by a dedicated skill described in Table 5.

As shown in Table 11, the agent correctly classifies the majority of suspicious test runs, while conservatively classifying one case as `UNCERTAIN`. All validation results are presented in Appendix A. The test run labeled as `UNCERTAIN` corresponds to ZK-4817, where the agent identifies the issue as a log-quality problem that degrades system diagnosability rather than a correctness bug. As a result, the agent does not classify it as a `TRUE_BUG`. However, since diagnosability is a critical aspect of distributed systems, we treat this case as an FSH failure in our evaluation. The average validation time is 120.74 seconds, which is significantly faster than manual inspection. Moreover, the average cost is approximately $0.50, making the approach economically practical.

Overall, agent-based failure validation can reduce manual effort in two ways: (1) by narrowing the focus to test runs labeled as `TRUE_BUG` and `UNCERTAIN`, and (2) by providing useful explanations that enable a quick overview of each test run.

Table 12. Runtime Overhead (seconds).

System	Baseline	Info Collection	Average Test Time
ZooKeeper	9.41	13.17	37.64
Kafka	7.12	34.89	44.86
HDFS	24.87	27.36	52.23

Table 13. Static Analysis Overhead.

System	Analysis Time (seconds)	Memeory consumption (MB)
ZooKeeper	11.02	624
Kafka	164.00	4694
HDFS	61.92	8016

6.4 Overhead Analysis

Table 12 shows the runtime overhead of Sieve. The **Baseline** column shows the original workload run time without any instrumentation. The **Info Collection** column shows the average fault-free run time of the instrumented system. The **Average Test Time** column shows the average run time with one fault injection.

The results show that Sieve introduces 1.1× to 4.9× baseline time for runtime information collection. On average, testing a fault point consumes around 1.9× to 6.3× baseline time. Each test run includes a series of operations such as initializing the execution environment, executing the workload, collecting runtime information, deciding the fault injection, and checking failure symptoms. Additionally, the static analysis completes within three minutes and consumes at most 8GB of memory for each cloud system shown in Table 13.

7 Discussions

Fault Model. Sieve currently focuses on injecting one delay to simulate the fail-slow hardware. However, the fail-slow hardware may incur other subtle faults such as exceptions and multiple delays. Hence, Sieve misses some bugs caused by these subtle faults. For example, in HDFS-11755 [29], the last datanode in the pipeline is affected by the fail-slow disk, leading to timeouts in other datanodes. When this last datanode receives the whole block, it sends the increased block report (IBR) to the namenode. After that, the fail-slow disk becomes fail-stop. The last datanode reports the transferred block to be missing. The namenode considers that the unconstructed block is lost. This is inconsistent with the specification of HDFS. To support various fault models, Sieve implements the exception injection and memory access tracing in cloud systems. However, existing fault injection strategies in Sieve are inefficient for other fault models. Therefore, in the future, we aim to design more fault injection strategies to adapt different fault models.

Fault Point Analysis. The keywords, e.g., `synchronized`, used in synchronized and timeout mechanisms are necessary for the fault point analysis in Section § 4.3. Sieve has analyzed several most common keywords: `synchronized`, `lock`, `wait`, `await`, `tryacquire`, `join`, `nanoTime` and `currentTimeMillis`. Our implementation is modular, which makes it convenient for developers to add more keywords and patterns completing the fault point analysis in Sieve.

Workload Driver. Table 10 indicates current workload drivers cannot trigger all candidate fault points since the used workloads are simple, which limit the code coverage. Fuzzing is an automatic

input generation technique [3, 7, 16, 23, 55, 66, 69, 70, 75] that can improve the code coverage. Developers can integrate fuzzing tools into Sieve to explore more candidate fault points.

Portability Porting Sieve involves different levels of effort depending on whether the target system uses a supported programming language. For a new Java-based cloud system, developers mainly need to provide system-specific build/deployment scripts, representative workloads, and optional system-specific checkers. Sieve already supports common Java I/O libraries and common synchronization and timeout patterns. The remaining manual effort is to register customized I/O APIs if the target system implements application-level I/O wrappers not covered by standard libraries. After these inputs are provided, call graph construction, fault point identification, bytecode instrumentation, and fault injection scheduling are automatic.

Although Sieve is currently implemented in Java, the core idea of Sieve can be applied in cloud systems implemented by other programming languages. Moreover, it is not difficult to implement Sieve in other programming languages. For example, there is a workflow of implementing Sieve in C++. Specifically, the approach of identifying synchronized and timeout-protected I/O points is similar. First, we need to find the keywords related to synchronized and timeout mechanisms. In C++, we use some keywords such as lock() in mutex and clock() in ctime. Second, we need to identify the I/O operations. In C++, we directly identify I/O operations in runtime libraries such as glibc. In this way, we avoid spending lots of time identifying different application-level functions that call the same I/O operations in runtime libraries. Third, we leverage the approach in Section § 4.3 to identify the I/O operations in synchronized and timeout mechanisms. Steps 1 and 2 only require manual effort once for different programming languages. Step 3 is automatic. The logic of the remaining components in Sieve is the same and can be easily implemented in C++.

8 Related Work

Fault Injection. In recent years, various fault injection techniques [1, 6, 18–20, 24, 36, 37, 47, 53] have been proposed to detect bugs triggered by various faults in cloud systems. Many existing schemes focus on coarse-grained faults and adopt various fault injection strategies. For example, CrashTuner [47] injects node crashes when meta-info variables are accessed; NEAT [1] provides simple APIs for developers to inject network partitions. Unlike them, Sieve instruments fail-slow agents before I/O operations to enable fine-grained FSH fault injection. Recent fine-grained delay injection tools [8, 68] fail to efficiently explore the FSH fault injection space due to overlooking the characteristics of FSH failures. Our work provides a bug study to analyze FSH failures and concludes some findings shown in § 3 Inspired by our bug study, Sieve is proposed to efficiently explore the FSH fault injection space.

CORDS [17] and WASABI [61] inject expections into cloud systems to detect error handling bugs. However, Sieve injects delays to detect FSH bugs. Sieve is orthogonal to CORDS and WASABI.

Distributed System Model Checker. Distributed system model checkers [39, 50, 72] intercept non-deterministic distributed events and permute their ordering. These schemes detect protocol bugs caused by complex interleaving of node-level events, such as message and node crash/reboot, while Sieve aims to detect implementation-level bugs triggered by fine-grained FSH faults. Sieve is complementary to existing schemes.

Distributed Concurrency Bug Detection. Several schemes [41, 43, 46, 74] have been proposed to detect concurrency bugs in cloud systems. For example, DCatch [41] uses specific happen-before rules to identify conflicting operations and reorders them. Sieve is an FSH fault injection framework that aims to detect diverse bugs including concurrency bugs.

Bug Studies on Fail-Slow Incidents. Guanwai et al. [25] present a study of fail-slow hardwares and analyze how fail-slow hardwares occur. Several works [44, 48, 57] provide studies of fail-slow failures at the software level and analyze how to accurately detect fail-slow failures by in-production

monitoring. However, unlike them, our bug study aims to study how fail-slow hardwares affect the software so that we can design an effective and efficient fault injection tool to detect FSH bugs before releasing production. Besides, our bug study shows that FSH failures caused by fail-slow hardwares include fail-slow and fail-stop failures. Our bug study is complementary to existing bug studies.

Fail-Slow Failure Detection. Several works [33, 44, 48, 57] develop advanced detectors for fail-slow failures. Panorama [33] detects fail-slow failures by enhancing the observability of different system components. IASO [57] detects fail-slow nodes based on timeout signals and peer evaluation. OmegaGen [44] generates customized watchdogs for detecting and localizing fail-slow failures. PERSEUS [48] leverages machine learning techniques to detect fail-slow failures. These detectors focus on in-production monitoring and require a significantly long time to detect failures, which are not suitable for testing before releasing production. For example, ISAO has been deployed for more than 1.5 years to catch fail-slow failures. However, Sieve focuses on detecting FSH bugs before releasing production by injecting FSH faults.

9 Conclusion

Fail-slow hardwares cause severe failures in cloud systems. This work presents a study of 48 real-world FSH failures in cloud systems to analyze their characteristics. We propose Sieve, a novel fault injection testing framework that enables fine-grained FSH fault injection for detecting FSH bugs. Sieve treats synchronized and timeout-protected I/O operations as candidate fault points that are likely to trigger FSH bugs. Sieve implements grouping and context-sensitive injection strategies to efficiently explore candidate fault points. Our evaluation shows that Sieve is effective and efficient in detecting FSH bugs in real-world cloud system.

Acknowledgments

We are grateful to anonymous reviewers for their comments and suggestions.

References

[1] 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.

[2] HDFS architecture. [n. d.]. Online. Available: https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

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

[4] Apache Cassandra. [n. d.]. Online. Available: https://cassandra.apache.org.

[5] K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

[6] 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.

[7] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 197–208.

[8] 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.

[9] ClaudeCode. [n. d.]. Online. Available: https://claude.com/product/claude-code.

[10] Codex. [n. d.]. Online. Available: https://github.com/openai/codex.

[11] Google compute engine incident 17008. [n. d.]. Online. Available: https://status.cloud.google.com/incident/compute/17008.

[12] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. 2018. Understanding real-world timeout problems in cloud server systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 1–11.

[13] D.Nadolny. 2016. Debugging distributed systems.. In *SREcon*.

[14] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. 2013. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–14.
[15] Gen Dong, Yu Hua, Yongle Zhang, Zhangyu Chen, and Menglei Chen. 2025. Understanding and Detecting {Fail-Slow} Hardware Failure Bugs in Cloud Systems. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 1127–1142.
[16] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 679–696.
[17] 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.
[18] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An empirical study on crash recovery bugs in large-scale distributed systems. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 539–550.
[19] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *Proceedings of IEEE/ACM SIGSOFT International Conference on Software Engineering (ICSE)*.
[20] Yu Gao, Dong Wang, Qianwang Dai, Wensheng Dou, and Jun Wei. 2022. Common data guided crash injection for cloud systems. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 36–40.
[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 29–43.
[22] Gocardless:API and Dashboard outage on 10 October 2017. [n. d.]. Online. Available: https://aws.amazon.com/cn/message/12721/.
[23] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation*. 206–215.
[24] 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)*. 238–252.
[25] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-slow at scale: Evidence of hardware performance faults in large production systems. *ACM Transactions on Storage (TOS)* 14, 3 (2018), 1–26.
[26] Per Brinch Hansen. 2022. The evolution of operating systems. In *Classic operating systems: from batch processing to distributed systems*. Springer, 1–34.
[27] Apache HBase. [n. d.]. Online. Available: https://hbase.apache.org.
[28] HBase-26195. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/HBASE-26195.
[29] HDFS-11755. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/HDFS-11755.
[30] HDFS-5341. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/HDFS-5341.
[31] HDFS-5522. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/HDFS-5522.
[32] HDFS-9178. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/HDFS-9178.
[33] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 1–16.
[34] INS. [n. d.]. Online. Available: https://www.instagram.com/.
[35] Javassist. 1999. Online. Available: https://www.javassist.org/.
[36] Jepsen. 2016. Online. Available: https://github.com/jepsen-io/jepsen.
[37] 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.
[38] Apache Kafka. [n. d.]. Online. Available: https://kafka.apache.org.
[39] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 399–414.
[40] 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.

[41] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. 2017. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 677–691.
[42] Haopeng Liu, Shan Lu, Madan Musuvathi, and Suman Nath. 2019. What bugs cause production cloud incidents?. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 155–162.
[43] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically detecting time-of-fault bugs in cloud systems. *ACM SIGPLAN Notices* 53, 2 (2018), 419–431.
[44] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 559–574.
[45] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 295–310.
[46] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. 2018. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 3–14.
[47] 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.
[48] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. 2023. Perseus: A Fail-Slow Detection Framework for Cloud Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 49–64.
[49] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. 2022. NVMe SSD failures in the field: the Fail-Stop and the Fail-Slow. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 1005–1020.
[50] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, et al. 2019. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.
[51] Ao Ma, Rachel Traylor, Fred Douglis, Mark Chamness, Guanlin Lu, Darren Sawyer, Surendar Chandra, and Windsor Hsu. 2015. RAIDShield: characterizing, monitoring, and proactively protecting against disk failures. *ACM Transactions on Storage (TOS)* 11, 4 (2015), 1–28.
[52] MapReduce-1800. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/MAPREDUCE-1800.
[53] Chaos Monkey. 2012. Online. Available: https://netflix.github.io/chaosmonkey.
[54] opencode. [n. d.]. Online. Available: https://github.com/anomalyco/opencode.
[55] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.
[56] PagerDuty. [n. d.]. Online. Available: https://www.pagerduty.com/.
[57] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. 2019. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 47–62.
[58] Daniel J Scales, Mike Nelson, and Ganesh Venkitachalam. 2010. The design of a practical system for fault-tolerant virtual machines. *ACM SIGOPS Operating Systems Review* 44, 4 (2010), 30–39.
[59] AWS service disruption. [n. d.]. Online. Available: https://aws.amazon.com/cn/message/12721/.
[60] Jira Software. [n. d.]. Online. Available: https://www.atlassian.com/software/jira.
[61] Bogdan Alexandru Stoica, Utsav Sethi, Yiming Su, Cyrus Zhou, Shan Lu, Jonathan Mace, Madanlal Musuvathi, and Suman Nath. 2024. If At First You Don't Succeed, Try, Try, Again...? Insights and LLM-informed Tooling for Detecting Retry Bugs in Software Systems. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. 63–78.
[62] Jeff Terrace and Michael J Freedman. 2009. Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*.
[63] TikTok. [n. d.]. Online. Available: https://www.tiktok.com/.
[64] MapReduce tutorial. [n. d.]. Online. Available: https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.
[65] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[66] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. 579–594.
[67] WeChat. [n. d.]. Online. Available: https://weixin.qq.com/.

[68] 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.
[69] Mingyuan Wu, Kunqiu Chen, Qi Luo, Jiahong Xiang, Ji Qi, Junjie Chen, Heming Cui, and Yuqun Zhang. 2023. Enhancing Coverage-Guided Fuzzing via Phantom Program. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1037–1049.
[70] Mingyuan Wu, Yicheng Ouyang, Minghai Lu, Junjie Chen, Yingquan Zhao, Heming Cui, Guowei Yang, and Yuqun Zhang. 2023. SJFuzz: Seed and Mutator Scheduling for JVM Fuzzing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1062–1074.
[71] X. [n. d.]. Online. Available: https://x.com/.
[72] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI'09*. 213–228.
[73] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed {Data-Intensive} systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.
[74] Xinhao Yuan and Junfeng Yang. 2020. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1141–1156.
[75] M. Zalewski. 1999. American Fuzzy Lop. Online. Available: https://lcamtuf.coredump.cx/afl/.
[76] Apache ZooKeeper. [n. d.]. Online. Available: https://zookeeper.apache.org.
[77] ZooKeeper-4074. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-4074.
[78] ZooKeeper-710. [n. d.]. Online. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-710.

A The results of Agent-based Failure Validation

Table 14. The results of Agent-based Failure Validation

FALSE POSITIVE CASE 1
Verdict:FALSE_POSITIVE
Justification:
A 30s fail-slow disk on leader sys3 (Snapshot taken in 30015 ms, system3.log:1083) caused followers to timeout after initLimit*tickTime=20s and re-elect sys2 as new leader. The stale leader sys3 subsequently detected loss of quorum via the designed 'waitForEpochAck' timeout (system3.log:1227-1231), stepped down, rejoined as follower, and the cluster fully synchronized by 02:50:23 with no data loss or inconsistency. The ERROR-level LearnerHandler EOFException and LeaderConnector 'retries exceeded' messages are expected artifacts of ZooKeeper's documented leader-election/recovery flow (per zookeeperInternals: 'leader and remaining followers will timeout and go back to leader election'). All observed symptoms are explained by ZooKeeper's intended fault-tolerance mechanisms; no FSH symptom persists beyond the fault, and consistency guarantees hold.

ZK-4817
Verdict: UNCERTAIN
Justification:

The CancelledKeyException (session 0x0) is a WARN-level, already-caught exception on a pre-established client connection, produced because the 20s doIO stall exceeded the client's handshake timeout. The cluster continues normal operation in both runs (sessions still committed, follower stays in broadcast), no FSH symptom (node unavailability, data loss/inconsistency, crash, stuck client, persistent perf degradation) manifests, and the behavior falls entirely within ZooKeeper's designed timeout-and-recovery path. The presence/absence of the exception between the 20s and 25s runs is timing-dependent, not a correctness divergence. However, the log message 'CancelledKeyException causing close of session: 0x0' is misleading client suggests client-initiated disconnect while the true trigger is server-side fail-slow I/O constituting a diagnosability issue (poor error reporting), which fits the UNCERTAIN tier rather than TRUE_BUG or FALSE_POSITIVE.

ZK-4836
Verdict: TRUE_BUG
Justification:
System3 received a SNAP from the leader containing only aclIndex=1 while the leader's ACL cache already held entries 2 and 3 (system3.log:202-204). After joining broadcast, getACL on /2 fails deterministically with 'ACL not available for long 3' and RuntimeException in ReferenceCountedACLCache.convertLong (system3.log:227-238). This is a persistent Data Inconsistency causing Node Service Unavailability for ACL-referencing paths; it violates ZooKeeper's snapshot-sync invariant that every ACL id referenced in the DataTree must exist in the ReferenceCountedACLCache, is not recovered by quorum retries/timeouts, and is consistent with a fail-slow fault injected during snapshot serialization/transmission.

ZK-4844
Verdict: TRUE_BUG
Justification:
A fine-grained fail-slow disk injected into writeLongToFile during setAcceptedEpoch/registerWithLeader blocks the follower's QuorumPeer thread synchronously inside AtomicFileWritingIdiom (system2.log:262-278, systemAsync1.log:247-263). Throughout the 60s slowdown the follower is alive but its ZooKeeperServer is not running, so every client session is closed with 'ZooKeeperServer not running' Node Service Unavailable (3.1) and Client Stuck (3.6). The failure persists for the entire fault duration; no node-level timeout or watchdog interrupts the stuck disk write, and the follower emits no diagnostic that points to the slow disk. The zookeeper.learner.asyncSending=true option only governs Learner packet sending and is orthogonal to epoch-file persistence, so the async run exhibits the same 60s hang (systemAsync1.log:579-590). Recovery only occurs after the fault clears, triggered externally by the leader's initLimit-based disconnect (EOFException at Learner.syncWithLeader), confirming there is no follower-local fault-tolerance mechanism for fail-slow disks on this safety-critical path.

FALSE POSITIVE CASE 2
Verdict: FALSE_POSITIVE
Justification:

The 60s follower hang observed in the SYNC logs reproduces the exact scenario documented in the ZooKeeper Administrator's Guide entry for learner.asyncSending (linking ZOOKEEPER-3575 / ZOOKEEPER-4074): synchronous learner packet sending in a critical section can hang on slow I/O. ZooKeeper ships a supported, documented mitigation — learner.asyncSending=true — which moves sending to a dedicated LearnerSender thread. The systemAsync1-3.log runs empirically confirm that with this option enabled the injected 60 s fail-slow fault no longer blocks the follower's learner critical section, and the cluster behaves as designed. Because a documented asynchronous option resolves the problem, the sync-mode hang is a known, configurable design limitation rather than an unhandled FSH bug. Additionally, quorum (2/3) is preserved throughout the fault, no data loss or inconsistency is observed, and the affected follower rejoins normally after the fault ends. Per Section 5 criterion 2, the behavior is explained by a designed fault-recovery mechanism.

FALSE POSITIVE CASE 3
Verdict: FALSE_POSITIVE
Justification:
A 60 s hang was injected into the synchronous Learner.readPacket on follower myid=1. Only that follower was temporarily unable to serve clients; leader (myid=2) and follower (myid=3) preserved quorum and continued committing writes, and the affected follower auto-recovered via snapshot sync. ZooKeeper's design explicitly tolerates one slow node in a 3-node ensemble and provides automatic client reconnect to other servers. Furthermore, the async option 'zookeeper.learner.asyncSending=true' (disabled in this run) is designed to tolerate exactly this kind of blocking learner I/O. The behavior is fully explained by designed fault-recovery mechanisms and an available asynchronous option, so it does not constitute a real FSH bug.

FALSE POSITIVE CASE 4
Verdict: FALSE_POSITIVE
Justification:
The Leader sid=2 experienced a fail-slow disk during initial empty-DB snapshot (Snapshot taken in 30024 ms). This caused transient 'ZooKeeperServer not running' responses and one follower (sid=3) re-election via a 20s Learner.registerWithLeader read timeout. All observed effects (SocketTimeoutException, EOFException in LearnerHandler, re-election, DIFF resync with zxid=0x0, NEWLEADER/UPTODATE) are designed Zab fault-recovery behaviors. After the snapshot completed at 02:19:31,843, quorum formed with sids [2,3], followers transitioned to broadcast, and transactions (0x100000001+) and session commits proceeded normally on all three nodes. The 'Got zxid 0x100000001 expected 0x1' message is a known benign Follower@169 warning on the first txn of a new epoch. No data loss, inconsistency, crash, or sustained unavailability beyond the fault duration was observed, and the checker trajectory.json is empty (no oracle violation).

ZK-4816
Verdict: TRUE_BUG
Justification:

A 30-second fail-slow I/O delay injected into the leader's synchronous snapshot serialization (SnapStream.sealStream) during Leader.lead() blocked the leader from completing initialization and responding to followers. This caused: (1) Node Service Unavailable - the entire cluster could not serve clients for 51 seconds; (2) a dual-leader condition where Node 3 (round 1) and Node 2 (round 2) simultaneously believed they were leaders; (3) the failure escaped ZooKeeper's heartbeat-based fault detection because the leader process remained alive but blocked on slow I/O. The synchronous snapshot path has no asynchronous alternative, and normal timeout handling could not recover the affected leader node until the fail-slow fault resolved. The cluster only recovered after the 30s delay completed, confirming the symptoms persisted for the duration of the fail-slow condition.

FALSE POSITIVE CASE 5
Verdict: FALSE_POSITIVE
Justification:
The 20s fail-slow injection on follower myid=1's NIOServerCnxn.doIO exceeds the 15s client session timeout, so the leader's expiration of session 0x2004307128b0000 is exactly the documented ZooKeeper behavior (Programmer's Guide: Sessions). No node crashed, quorum was maintained, persistent data was intact, and a new client session recovered immediately. Observed -101/-102/-110 codes are normal application-level return values, and the 'Got zxid 0x100000001 expected 0x1' WARN is a known benign first-epoch log. No FSH symptom is triggered beyond what the design already tolerates.

FALSE POSITIVE CASE 6
Verdict: FALSE_POSITIVE
Justification:
Node 3 (leader) experienced a 20s fail-slow disk on writeLongToFile while entering broadcast phase. Followers 1 and 2 timed out per ZAB's initLimit/syncLimit (10s), elected Node 2 as new leader (epoch 0x2), and resumed service. The old leader's EOFException ERROR logs are expected artifacts of followers closing sockets; no data loss, no inconsistency, no persistent unavailability. Node 3 rejoined as follower via snapshot sync (zxid advanced to 0x2000000ab). The cluster recovered via ZooKeeper's designed leader-election fault tolerance, well before the fail-slow fault was resolved, so criterion #2 (not explained by designed recovery) and #3 (symptoms persist until fault cleared) in Section 5 are not satisfied.