Demystifying and Checking Silent Semantic Violations in Large Distributed Systems
Chang Lou Yuzhuo Jing Peng Huang
Johns Hopkins University

Abstract
Distributed systems today offer rich features with numerous semantics that users depend on. Bugs can cause a system to silently violate its semantics without apparent anomalies. Such silent violations cause prolonged damage and are difficult to address. Yet, this problem is under-investigated.

In this paper, we first study 109 real-world silent semantic failures from nine widely-used distributed systems to shed some light on this difficult problem. Our study reveals more than a dozen informative findings. For example, it shows that surprisingly the majority of the studied failures were violating semantics that existed since the system’s first stable release.

Guided by insights from our study, we design Oathkeeper, a tool that automatically infers semantic rules from past failures and enforces the rules at runtime to detect new failures. Evaluation shows that the inferred rules detect newer violations, and Oathkeeper only incurs 1.27% overhead.

1 Introduction
Users’ increasing reliance on distributed systems highlights the importance of ensuring they work correctly. Unfortunately, real-world distributed systems inevitably encounter failures. When a failure is recognizable through explicit signals such as crash, timeout, error code, or exception, timely actions can still be taken to detect and mitigate the failure. A vexing problem occurs when a system is operational but silently breaks its semantics without apparent anomalies.

Take a distributed notification service as an example, which provides an interface that promises to invoke the client callback whenever the status of some object changes. A bug may cause this system to miss invoking the callback upon a change or invoke the callback more than necessary. A vexing problem occurs when a system is operational but silently breaks its semantics without apparent anomalies.

Table 1: Number of public interfaces in popular distributed systems.
An interface can have multiple semantics under different settings.

<table>
<thead>
<tr>
<th>System</th>
<th>Ver.</th>
<th>Client API</th>
<th>Public Method</th>
<th>Admin Command</th>
<th>Config</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZooKeeper</td>
<td>3.4.6</td>
<td>38</td>
<td>219</td>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td>ZooKeeper</td>
<td>3.6.2</td>
<td>78</td>
<td>2,853</td>
<td>18</td>
<td>128</td>
</tr>
<tr>
<td>HDFS</td>
<td>2.7.2</td>
<td>128</td>
<td>5,293</td>
<td>11</td>
<td>224</td>
</tr>
<tr>
<td>HDFS</td>
<td>2.10.0</td>
<td>162</td>
<td>6,306</td>
<td>12</td>
<td>449</td>
</tr>
<tr>
<td>Kafka</td>
<td>2.6.0</td>
<td>166</td>
<td>2,661</td>
<td>76</td>
<td>366</td>
</tr>
<tr>
<td>Kafka</td>
<td>2.8.0</td>
<td>171</td>
<td>3,107</td>
<td>86</td>
<td>379</td>
</tr>
</tbody>
</table>

Distributed systems today have rich semantics (Table 1) exposed through client APIs, public methods including RPCs among internal components, administrator commands, configuration parameters, etc. One interface often encodes multiple guarantees. New interfaces and semantics are also continuously introduced as a system evolves. These characteristics together make it challenging to ensure that a distributed system conforms to its semantics in production settings.

Indeed, real-world evidence shows that semantic violations occur in practice. In a Google cloud incident [3], a traffic engineering subsystem that is supposed to throttle traffic upon congestion incorrectly throttled traffic even though the network was not congested. Another highly-impactful global outage [2] was caused by a quota system incorrectly reporting the usage for a user ID service as zero.

However, other than anecdotal evidence, the problem of silent semantic violations in distributed systems remains mysterious, despite its severe consequences. For instance, mature distributed systems include extensive test cases to check the correctness of their features. Thus, it is natural to assume silent semantic violations are rare in production because testing likely has eliminated most of them. In addition, while adding assertions and runtime verification [43, 44, 48, 57] are potential solutions, the conventional wisdom is that they are expensive and semantic rules are difficult to get. It is also unclear what kind of semantics are violated in practice.

To systematically understand this problem, we present, to our best knowledge, the first empirical study on 109 real-world silent semantic violations from nine widely-used distributed systems. Through these cases, we analyze key questions such as how prevalent are semantic violations in practice, what semantics are violated, why are these failures not caught in testing, and how are these silent violations detected.

Our study provides quantitative data points to answer these
questions. The study findings also challenge some conventional wisdom and reveal gaps in the current practice. We highlight several findings:

• Contrary to the belief that silent semantic violations rarely occur in deployed systems, they have significant presence (39%) among sampled failures of all kinds.
• While the studied systems get more extensively tested over time and continue to add new features and semantics, their initial semantics do not become more bulletproof. On the contrary, more than two thirds of the failures violate semantics that have existed since the system’s first stable release.
• Although these are distributed system failures, most (74%) violations can be determined locally in some component.
• The violated semantics are often not untested but rather well covered by existing test cases.
• Enabling assertions in release builds helps by converting semantic violations into crash failures. One studied system does this and has the lowest ratio of semantic failures.
• In many cases, although a semantic was initially honored, it was later violated, thus one-time assertions are insufficient.
• Many system semantics are vulnerable to violations during maintenance operations or node events.

Given the prevalence (as our study indicates) and severity of silent semantic violations, we design a tool Oathkeeper to help users check silent semantic violations at runtime. The tool design is directly guided by insights from our study.

Specifically, we find that in 73% of the cases, developers add regression tests after the failure is reported, which contain valuable information about the failed semantic. However, the majority of the studied cases still violate semantics that have been tested before. A major reason for the gap is that these regression tests are usually patch-driven: they only check if the specific bug is fixed in a particular setup using a bug-triggering workload. The underlying semantics can continue to be broken with different root causes in different scenarios.

Based on this insight, Oathkeeper leverages the regression tests and tries to infer the underlying semantic rules implied by the tests. To do so, Oathkeeper runs the tests on both the buggy version and patched version of the system, and takes a template-driven approach to automatically infer semantic rules from the two traces. Oathkeeper then deploys these semantic rules to production to catch future violations that are caused by different bugs under different conditions.

We evaluate Oathkeeper on ZooKeeper, HDFS, and Kafka. Oathkeeper infers hundreds to thousands of semantic rules from the old regression tests in these systems. With the inferred rules, we evaluate Oathkeeper on seven real-world semantic failures that were introduced long (9–34 months) after the old failures. Oathkeeper detects violations for six of them. With all rules enabled, Oathkeeper on average only incurs 1.27% throughput overhead to the target systems.

The contributions of this paper are two-fold: (i) the first study on real-world silent semantic violations in nine popular distributed systems; (ii) the design of Oathkeeper, which automatically infers semantic rules for large distributed systems to check silent semantic violations at runtime.

The source code of Oathkeeper is publicly available at: https://github.com/OrderLab/OathKeeper

2 Background

2.1 Definition

We consider a distributed system $S$ that provides services through a collection of operations. Each operation $o$ has certain semantics [29]. The semantics encode guarantees that $o$ makes about the output, system states, and results of subsequent operations, in response to some triggering condition $c$. The condition $c$ can be a client request, an admin command (at the server side), a message from internal components, as well as an environment change including the passage of time. The semantics of $S$ are all the guarantees provided by the history of operations $S$ executes in response to a list of $c$.

A semantic violation (failure) occurs when $S$ breaks some of its semantics in an execution. The failures may exhibit explicit error signals, such as crashes, timeouts, and exceptions. In such cases, the violations overlap largely with existing failure models and can be well addressed by existing techniques.

This work focuses on silent semantic violations, in which $S$ violates its semantics but remains operational without exhibiting explicit error signals ($S$ is unaware of its misbehavior). We focus on this class of failures because they are under-studied yet incur damaging consequences, and they pose significant challenges to testing, failure detection, and recovery.

Silent semantic violations differ from other failure modes in observability. Fail-stop failures cause complete loss of functionality, which can be observed with simple measures such as monitoring heartbeats. Fail-slow [32], partial failures [46] and gray failures [37] only cause some functionality to be broken (slow). But these issues can still be observed with generic approaches, e.g., checking exceptions or timeouts [45]. In comparison, silent semantic violations are difficult to observe without a deep understanding of $S$’ semantics and execution.

Another way to interpret the “silent” aspect is on the semantics being violated. If $S$ only has a few operations, all of which have well-defined and thoroughly checked semantics, semantic violations in $S$ will be observable failures. Unfortunately, distributed systems have a large number of interfaces (Table 1), many of which have loosely-defined (or hidden) semantics that cannot be easily checked. Consequently, violations of such semantics are difficult to detect and address.

2.2 An Example

We show an example of silent semantic failures from our study (Section 3). ZooKeeper is a coordination service with a hierarchical data model. Its clients store data by creating $znode$ in a namespace. A special type of $znode$ is called ephemeral node. The semantics of the ephemeral node $create()$ operation guarantees that the $znode$ exists for as long as the creating
client’s session and will be deleted once the associated session ends. The triggering conditions are the create request and the client session disconnection. Ephemeral nodes are commonly used to store membership information. For example, HDFS implements its leader election using ephemeral nodes [27].

In a production ZooKeeper failure [13], some ephemeral node still existed even though the client session that created them was long gone. Specifically, a Kafka consumer crashed but the associated znode was not deleted (Figure 1). As a result, when Kafka clients queried ZooKeeper to discover consumer information, they kept trying to connect to the crashed consumer. In other settings, this semantic violation can propagate to other dependent applications, e.g., it will break HDFS namenode’s automatic fail-over feature, which depends on the ephemeral node semantics, causing an HDFS service outage.

3 Study Methodology

Compared to other failure modes in distributed systems, silent semantic violations are not well understood. To fill this gap, we conduct a study on user-reported silent semantic failures from nine large-scale distributed systems (Table 2). We select these systems because they are representative, mature, widely used in production, and record many user-reported failures.

To collect the failure cases, we first query the study systems’ issue trackers to find tickets that (1) are marked as “bugs”, (2) have priorities higher than “minor”, (3) are resolved, (4) involve the server components. This step returns a large number of tickets. We then randomly sample a subset (Table 2). Among this subset, some are not real failures, such as issues found in internal testing. The remaining ones (valid column in Table 2) are potential production failures. We then read their descriptions and check whether the failures violate system semantics. We filter crashes, aborts, out-of-memory errors, and semantic failures with clear error signals.

After the above step, we get a candidate set of production silent semantic failures (Candidate column). Due to time constraints, we perform in-depth analyses on a subset of the candidate cases, preferring those with sufficient information and discussions. This gives us the final study dataset (Studied column) of 109 production semantic failure cases.

Note that our sample sizes vary across systems. This is because the studied systems’ tickets vary greatly in terms of their information, quality, and bug types. If using a fixed sample size or ratio, one system can dominate the study and produce extremely biased findings. Our sampling instead is done iteratively: for a particular system, if after an initial sampling, its number of Candidate cases is too small or 0, we sample more, until the candidate numbers for different systems are relatively balanced. Note that each iteration in this process is still randomly choosing from the All tickets.

Threats to Validity. Like all empirical studies, our study is subject to validity problems such as the representativeness and biases. We cover popular distributed systems of different types, such as database, file system, and search engine, to improve the representativeness. To minimize selection bias, we randomly sample the cases. We also spread the sampling across times so we are not biased by some specific version. To reduce the manual inspection errors, we write a detailed analysis document for each case and have multiple inspectors examine each document to reach a consensus.

Although our study provides informative findings on semantic failures in the studied systems, they may not be generalized to other systems beyond the scope this study was conducted. Our study is also biased by programming languages (Java and C++); the findings may not generalize to systems written in other languages such as Erlang or Elixir, which embrace “let-it-crash” error handling philosophy [18].

4 Are Silent Semantic Failures Rare?

Prevalence. An important question about silent semantic violations is whether they occur rarely in production. Getting accurate prevalence data requires examining thousands of tickets for each system, which is a daunting task. We instead obtain an approximate result by calculating the percentage of silent semantic failures in our sample set. Specifically, we calculate the percentages of the number of candidate cases in Table 2 over the number of valid cases in the sample. Note that the candidate cases are examined to be indeed silent semantic failures, even though we only study a subset of them.

Finding 1: Silent semantic failures have significant presence across all studied systems, occupying 20%–57% (39% on average) of the sampled cases for all types of failures.

The percentages vary in different systems. Systems such as ElasticSearch and Cassandra have a higher percentage of semantic failures (57% and 46%, respectively). MongoDB has the lowest ratio (20%). We will discuss in Section 8 these
systems’ practices that may contribute to the differences.

Severity. How severe are these reported silent semantic failures? To answer this question, we analyze the severity levels that developers assign to the issues. Some systems use slightly different categories. We normalize them into three levels: Blocker, Critical, Major. Based on the official descriptions, Blocker means the issue “should block release until it is resolved”; Critical means the issue causes severe consequences like data loss; Major means a “major loss of function”.

Overall, 45% of the studied cases have Blocker or Critical priorities. The ZooKeeper failure [13] described in Section 2.2 is an example Blocker issue. As another example Blocker issue, users reported that in their HDFS deployment, all the replicas of some blocks are residing on the same rack [8], which breaks the redundancy policy. This is clearly a severe violation because replica placement is critical to HDFS data.

We also compare the priority distribution of semantic failures with all failures in the sample. The result is shown in Figure 2. The average percentage of Blocker priority in semantic failures increases from 15% to 21%, and the percentage of Critical priority increases from 8% to 24%.

Interestingly, we find in some cases initially developers may not consider the symptoms to be severe, but after further investigation developers upgrade the priority level, e.g., “Marking as critical for 2.0. These ‘unexpected behaviors’ cause operator head-scratching and wasted hours of digging” [5].

Finding 2: Despite the lack of explicit error symptoms, silent semantic failures are considered severe by developers and users. Moreover, the sampled semantic failures are assigned with higher priorities compared to all sampled failures.

Consequence. We next analyze the failure consequences. Figure 3 shows that besides incorrectness, semantic failures cause serious consequences such as corruption and data loss.

The consequences are damaging because clients or users are misled by the system’s seemingly normal reactions. For example, Kafka guarantees that when a success response is sent to a producer, the produced message will be persisted by at least min_isr replicas. Otherwise, the producer will be notified of an error, so it may retry the request. In one failure [9], a leader replica switched to follower then back to leader. Some messages produced were lost while the client received responses with no error. This false success resulted in data loss for the users.

Note that Figure 3 is about the reported impact of failures, which is not always the semantic violation per se. For example, in a MongoDB case, the maximum cache usage configuration is not enforced. It takes a while for the violation to cause a performance problem—which is the consequence of this failure. But even before the system reaches the performance collapse, a cache limit violation has occurred.

Finding 3: In addition to incorrectness (wrong responses), silent semantic violations often cause severe consequences including corrupt state, data or state loss, and security issues.

5 What Kind of Semantics Is Violated?
5.1 Sources of Violated Semantics

The studied failures violate various system-specific semantics. We analyze where these semantics come from. There are four sources and Figure 4 shows their distributions:

- **API spec**: a system API promises certain effect will (not) occur, e.g., a successful return of removeWatch API is supposed to remove the specified watcher.
- **Internal behavior**: the system’s documentation explicitly guarantees that something should (not) occur about its internal behavior, which is not directly exposed to external APIs, e.g., HDFS guarantees that if some Erasure Coding blocks fail, they should be detected and reconstructed.
- **User configuration**: user configurations regulate some system behaviors and the guarantees depend on the user settings. For example, the max_hint_window_in_ms parameter in Cassandra defines the maximum time window the coordinator will generate hints for a dead host.
- **Implicit**: the semantics are not explicitly defined or documented, but users expect them to hold for a correct system.

Finding 4: Most (87%) studied failures violate semantics that are explicitly defined in API specs, system docs, or configs.

Interestingly, in 10% of the studied cases, the system does not respect its configuration’s semantics. For example, if users set acl.inheritance to true, HDFS should enable ACL inheritance; but in one case the inherited ACL permissions are masked [7]. This violation causes security issues. The problem of misconfiguration is extensively researched [20, 21, 35, 56]. This finding suggests that even when users set configuration properly, a system can still misbehave.
As an example of Implicit semantics, in one HBase case [4], a region is online in server A, but the region location registered in the meta table is server B. While this consistency semantics is a common sense, it is not explicitly declared.

Explicit documentation of semantics is indicative of developers’ awareness of its guarantees and importance. One hypothesis is that if the semantics in a failure is not documented, it is understandable that developers did not make enough efforts to enforce the semantics. This finding disproves this hypothesis. However, the explicit documentations do not translate into fewer violations. One reason is that developers often document the semantics in a vague (e.g., “should produce correct results”) or incomplete way. A more fundamental gap is that the documentation is designed to be human-readable but not machine-checkable. For example, the semantics for ephemeral znode in ZooKeeper is documented clearly, but the system does not have any mechanism or tool to enforce this semantics in deployment.

Implications: Rich sources of documentation exist to leverage and judge semantic violations. Developers should move from documenting semantics in informal text to rigorously declare semantics that are mechanically checkable and enforceable.

5.2 Categorizations of Violated Semantics

Old vs. New Semantics Modern distributed systems often keep adding new features. For example, the number of client APIs in ZooKeeper increased from 38 in version 3.4.6 (2016) to 78 in version 3.6.2 (2020). Similarly, HDFS’ key APIs in Fs timeouts increased from 128 in version 2.7.2 (2016) to 162 in version 2.10.0 (2019), along with significant increases of semantics in other interfaces such as RPC methods.

Since around 90% of our studied failures occurred after more than two years since the software’s initial release (Figure 5), a natural hypothesis is that most of them violate some new semantics. We validate this hypothesis by analyzing the age of semantics in the studied failures. We define old semantics as ones that exist since the first major stable release of the system and others as new semantics.

Surprisingly, we find only less than one third (32%) of our studied failures violate relatively new semantics, while 68% of them violate old semantics. Old semantics usually represent the most fundamental functionalities the system provides since developers implement them first, and they usually undergo extensive testing already. However, our finding suggests that (1) even with new features added to the system, old semantics are still ones violated the most; (2) even with testing accumulating over the years, the reliability of old semantics is not necessarily higher in newer versions. Take ZooKeeper as an example. Its ephemeral znode interfaces and semantics have existed since the first major stable release (3.0.0) in October 2008 [1]. However, there are still production failures violating the guarantees of ephemeral znode reported by users even 10 years later [15].

We further investigate why old semantics still keep getting violated. There are three broad reasons: (1) new implementation is buggy, developers may optimize, refactor or refine the implementation of existing functionality, which contain bugs that break old semantics, e.g., a concurrency bug introduced in changing an implementation to be multi-threaded; (2) new feature adds buggy interactions, when some new feature is added, developers may extend existing module to interact with or support the new feature. For example, after HDFS introduces the encryption zone feature, it needs to extend the original snapshot file function and the new handling path is buggy [6]; (3) latent bugs are exposed, as the most basic semantics, these old semantics’ original implementations can be complex and contain latent bugs that can only be exposed in very specific scenarios. In one ZooKeeper failure [14], users find the ephemeral znodes are not deleted when the system time changes unexpectedly. This bug exists for 6 years before it is discovered, because neither the testing nor most deployments would exercise the system with the time change.

Note that we did not count the numbers of semantics in the study, either for new or old semantics. This is because even with explicit documentation such as API specs, determining how many semantics are there for a given API can be subjective, which depends on the granularity of semantics. Instead, we objectively judge if the specific semantics violated in a failure were introduced in the initial release or not.

Finding 5: 68% of the studied failures violate old semantics.

Implications: Instead of having the false hope that old semantics are reliable, developers should invest efforts to prevent semantic violation regressions.

Local vs. Distributed Semantics Since the study subjects are distributed systems, we analyze whether the semantic violations naturally require considering multiple distributed components. This question is important to the design of runtime verification techniques [43,44,48,57].

We find that indeed 26% of the semantic violations require global information to judge, e.g., whether the replica placement policy in HDFS is correctly enforced, or whether states in different Cassandra nodes match the consistency level.

However, interestingly, we find that the majority (74%) of the violations can be determined in a local scope. For example, appendTo in HDFS has the semantics of appending data to the end of a target file and making it persistent. A buggy node may fail to persist the new blocks or accidentally overwrite them. The violations can be determined in this node.

One reason is that a distributed system component often keeps local copies of states for other components. For instance,
even though ZooKeeper session is a global concept (a client connection to any follower or leader constitutes a session), each node has a copy of the alive session and node list. The semantics of ephemeral znode, which require knowledge of the session information, can thus be checked locally in a ZooKeeper node.

Current runtime verification solutions typically aggregate global states across all nodes to check property violations. Obtaining such global information can be both expensive and tricky, e.g., dealing with consistency issues in capturing distributed snapshots [23]. Our finding suggests it may be sufficient to use local checkers to expose many semantic failures.

Finding 6: The violations in semantic failures can be usually (74%) determined in the scope of a single component.

Implications: Employing local checkers can potentially expose many semantic violations.

Safety vs. Liveness Semantics Some failures break safety-related guarantees. For example, in Kafka, the maximum number of consumers in a group should not be larger than a configured limit, but users found more consumers joined the group.

In comparison, other semantics are liveness related. For example, ZooKeeper specifies that a container-type znode with no child znodes should eventually be deleted. Even when we observe some empty container node exists, it does not necessarily indicate this guarantee is violated because it might still hold some time later. Without context, one can interpret some safety guarantee, such as a correct response should be returned, to be involving liveness, because even if a response is not received, it could be still on the way. We refer to the system’s official documentation for making the distinction. If the documentation explicitly states that when an operation returns, something (e.g., a notification) will eventually happen, then a failure about its absence is a liveness violation.

It is generally challenging to check liveness properties [38], because there can be infinite possibilities in the execution that eventually produce the desired effect. Fortunately, we find most (86%) of our studied failures violate safety semantics.

Finding 7: 86% of the studied cases violate safety semantics.

Implications: There is usually a fixed time point to determine if a system has violated its semantics.

6 Why Do Silent Semantic Failures Occur?

We analyze what causes a system to break its semantics. We are interested in identifying potential common bug patterns in the root causes, which can inform the designs of bug finding tools to eliminate semantic failures before production.

Some semantic failures are caused by bugs such as memory error, data race, and integer overflow, which are well studied with many tools designed to detect them. We find only 12% of the cases are caused by such bugs. The remaining failures are caused by system-specific logic bugs including design flaws, which are difficult to be caught by bug detection tools.

An interesting finding is that even for failure cases that violate the same or related semantics, their root causes can be quite different. Take the ZooKeeper ephemeral znode as an example: (1) ZK-1208 is caused by a race condition: when ZooKeeper is handling the close session request, it deletes ephemeral znodes and then removes the session, in between a create operation causes new ephemeral znodes to be added; (2) In ZK-3144, the violations are caused by an incorrect order: during request processing, the lastProcessedZxid is updated before sessions are modified, so a snapshot may not include the change and the ephemeral node is not deleted after log replay; (3) In ZK-2355, the violations are caused by buggy error handling: follower fails while reading the proposal packet, but resetting lastProcessedZxid is missed in the error handler; (4) In ZK-2774, the system time of a server is changed unexpectedly, and session expiration codes rely the absolute system time, which causes the ephemeral znodes to persist after the client is disconnected for a long time.

Finding 8: Only 12% of the studied failures are caused by well-defined bugs such as race conditions, while most cases are caused by a wide variety of logic bugs. Even for failures violating the same semantics, the root causes are diverse.

Implications: It can be challenging to exploit code patterns to eliminate semantic violations through static bug detection.

7 How Are Semantic Failures Manifested?

Timing of Violation Understanding when semantics are violated can shed light on how to detect the violation.

As Figure 6 shows, some semantics only exist during the execution of its associated operation (at return point), e.g., read operation should return the latest data. We call them short-lived semantics. In comparison, some semantics exist even after its associated operation finishes, e.g., the specified file in create operation should be persisted and continue to be available after create returns. They often only cease to apply after some other event, e.g., until a delete operation on the same file is executed. We call them long-lived semantics.

Interestingly, we find that 67% of the cases violate long-lived semantics. This is partly because these semantics have a larger “vulnerability” window compared to short-lived semantics: a violation can occur anytime in its lifespan. ZooKeeper ephemeral znode and watches are such examples. Essentially, the system must maintain the promise for a long time.

We categorize the violation timing into four scenarios: at the end of short-lived semantics (1), e.g., wrong response, at the start (2) or in the middle (3) or near the end (4) for long-lived semantics. An example for (2) is in HDFS-12217, the snapshot operation did not capture all open files, which
violates the long-lived snapshot semantics since the beginning. An example for 3 is HDFS-9083: at the block creation time, the block placement policy is honored; but after some node failures, all replicas of the block reside on the same rack. 4 happens when the semantics should cease to apply but did not, e.g., ephemeral nodes should be removed when clients timeout. Figure 6 shows the distributions of the four scenarios.

Finding 9: Near two thirds of the studied cases violate some long-lived semantics. In 40% of the cases, the semantics are initially honored but are violated in the middle.

Implications: It is crucial to continuously monitor semantic guarantees, even after the initial semantic check passes.

Failure Triggering Conditions We further examine what triggers the semantic failures. Figure 7 shows the result.

Finding 10: More than half of the studied failures are triggered by specific requests, while 39% of the failures require particular timing to trigger. Semantic failures often (41%) only manifest themselves under multiple types of conditions.

HDFS-14514 is an example of semantic failure that requires multiple types of triggering conditions. The semantic violation (read out file size from snapshot is incorrect) can be only triggered when 1) snapshot.capture.openfiles is set; 2) create empty directory and encryption zone; 3) a client keeps a file open for write under the empty directory; 4) append several times; 5) perform a maintenance operation, snapshot.

We also find that in 23% of the cases, the triggering condition is certain system maintenance operation, such as compaction, cluster upgrade, node decommissioning. Such events do not occur frequently. They trigger semantic violations often because during the maintenance operation, the system execution enters a different mode, which exposes rare bugs.

Implications: The reliability of semantics is vulnerable to maintenance operations or node events. Operators and the system should check violations during and after such actions.

8 Current Practice for Semantic Failures

8.1 Testing

Since semantic violations concern functionality correctness, testing is responsible for catching them. The prevalence (Section 4) of many semantic failures in production seems to suggest a lack of testing. But that is not the case. The systems we study have extensive test cases—a median of 1309 test files. In addition, in 73% of the studied cases, the system has at least one test case covering the violated semantics.

Then why the studied failures are not exposed during testing? The earlier Finding 10 provides some clues. In many cases, even though there are related test cases, they lack some operations or arguments key to trigger the production failure. Even when the test cases have the proper operations and arguments, they only exercise the system under one timing, one configuration or normal scenarios, while the bugs are only triggered with unique timing, configuration, or node failures.

Are the failure triggering conditions so special that it is impossible for developers to foresee? Interestingly, we find that in many cases, similar triggering scenarios do exist in the test suite but they are not used in testing the violated feature.

Finding 11: Semantic violations occur not simply due to a lack of testing. The violated semantics are usually (73%) covered by some existing test. In more than half of the studied failures, similar triggering conditions exist in the test suite.

A fundamental gap is that developers tend to write tests driven by examples or fixes for a specific bug. Such tests are not expressive enough to preserve the underlying semantics and prevent regression. Consequently, developers spend repeated efforts to add tests. In HDFS-14514, the server reads snapshot file with incorrect length from encrypted zones. This exact semantics is already checked in an existing test case. If that test "copies" one line of test configuration dfsAdmin.createEncryptionZone(...) from other tests, the new bug will be triggered and exposed.

Implications: Coverage of semantics alone is insufficient. Developers should introduce variances in existing test cases. It is also useful to "copy" triggering conditions across tests. More fundamentally, developers should write more general tests for the semantic properties rather than specific examples.

8.2 Assertions

Assertions are a common method for catching logic bugs, which are major contributors to semantic failures (Section 6). They are typically only used in development and are turned off by default in release build for performance and stability.

Some of our studied systems use assertions in production: MongoDB has added many invariant checks since 2014 [11]. Interestingly, as Section 4 shows, MongoDB has the lowest ratio of semantic failures compared to other systems. While this practice may cause instability, e.g., some users got infrequent crashes due to invariant check failures after upgrading to new versions [12], developers still prefer to fix the underlying bugs rather than turning off assertions completely.

We observe two gaps in the current practice. First, most existing assertions are pre-condition checks on the sanity of function arguments. They are too low-level to catch semantic violations, which require checking system functionalities and usually the operation history (e.g., in checking consistency violations [48]). Second, existing assertions are usually only activated once during an operation, e.g., the entry of a function. But many semantics are long-lived (Section 7), which require continuous validation until the lifespan of semantics ends.
Finding 12: Although in 51% of the failures the buggy functions have some sanity checks, few (9%) cases can be potentially detected by adding proper sanity checks.

Implications: Enabling assertions helps reduce silent semantic violations. However, developers should add more semantic-level invariant checks besides sanity checks.

8.3 Observability

Since our studied failures are silent violations, how do users notice these subtle failures then? Understanding this question can reveal insights to improve the observability of semantic failures. We carefully examine the discussion threads in each ticket. In 34 cases, users mentioned their experience clearly.

For all of these cases, users discovered the issues through noticing something suspicious in some “side channels”. We categorize them into two types: (i) benign errors in other requests (32%); (ii) anomalies in logs, files, or performance of other tasks (68%). In HBASE-11654, users find out the violations by noticing splitting directories in /hbase/WALS/, which is “very strange” because “those logs should have been replayed and deleted”. In KAFKA-9137, users observe the failure by seeing an increase in eviction rate in the logs. In CASSANDRA-6527, users found tombstones appeared even though they never used delete for a column family.

It might seem that we can rely on users to manually detect system semantic failures. Note that there is a survival bias: our studied cases by definition are identified, but in practice silent semantic violations can be easily missed because (i) users do not monitor the systems $24 \times 7$; (ii) when they check, they may not inspect the proper signals. When users notice the failures, the damage may be already done. In CASSANDRA-6527, users commented: “Fortunately, we have noticed that quickly and canceled the migration. However, we were quite lucky.”

How to make semantic failures more observable? First, if a system API has no interaction with others, it is hard to judge its correctness based on a single piece of information. In practice users often use multiple related APIs to cross-compare results. In HBASE-15236, users observe the violations because `get` and `scan` return different sizes for the same bulkloaded hfiles. Second, current systems often do not expose enough information about theirs internal states, thus users have to ad-hocly infer whether a promise is obeyed or not. Existing error messages (e.g., a legitimate exception for another request) only focus on the current request, which is hard to link to the semantic violation in past correlated requests.

Finding 13: Semantic violations are currently observed from “side channels”: 32% from errors in other requests, 68% from anomalies in logs, files or performance of other tasks.

Implications: Designs of overlapping APIs improve observability of semantic violations. Systems should provide more admin APIs for convenient query of their internal states. Error messages should provide hints about past correlated requests.

9 Oathkeeper: A Semantic Violation Checker

Guided by our study, we build a tool Oathkeeper to check semantic violations for large-scale distributed systems.

9.1 Design Overview and Workflow

Oathkeeper takes a runtime approach to check semantic violations in production. This choice is motivated by our findings that semantic failures have diverse root causes (Finding 8) and often difficult to expose in testing due to complex triggering conditions (Finding 10).

Central to a runtime verification approach is what invariants to use. Existing solutions rely on users to write distributed assertions to check the correctness of distributed protocols [43, 44] or network functions [57]. In those scenarios, the semantics to check are limited and well-defined. But in our cases, the systems have abundant (Table 1) and loosely-defined semantics. Even for semantics that can be described in simple expressions informally, mapping them to the concrete checkable invariants in the complex systems code is hard. These factors make manual construction a daunting task.

Insight and Key Idea. The insight behind Oathkeeper is based on our finding that the majority of the studied failures violate old semantics (Finding 5) despite the decent coverage of testing (Finding 10). When a semantic failure occurs, developers usually add regression tests. But these tests only check if the specific bug is fixed in a specific setup, while the same semantics can be violated repeatedly in other scenarios.

Based on this insight, Oathkeeper leverages the existing regression tests developers write for past semantic failures and automatically extracts the essence—the violated semantic rules. Oathkeeper then enforces these rules at runtime to detect future semantic violations, which may be caused by different bugs under different conditions.

Input and Output. To apply Oathkeeper to a new system, users supply a system-wide configuration and a list of past semantic failure metadata. The former provides basic information about the system such as the compilation command and test directory, and optionally the classes to include for analysis. The latter metadata is provided in the form of git commit id (for version switching) and regression test name.

Oathkeeper outputs the likely semantic rules (Section 9.3). Prior runtime verification tools focus on invariants expressed as predicates among key state variables in a system such as lock id and lock mode. This representation alone can be insufficient or complex to express the semantics of large distributed systems. Instead, Oathkeeper focuses on rules that describe relations among semantics-related events, particularly operation invocations and state updates. Such an event relation rule is expressive to capture various semantics.

Workflow. Figure 8 shows the tool’s workflow. Oathkeeper operates in two stages. In the offline stage, Oathkeeper instruments the target systems to record major events (1). It then
exercises the system twice with the regression tests: once using the patched version and the second time using the buggy version. This will generate two sets of traces (2). The inference engine infers likely semantic rules from the traces of the patched version (3). The verifier applies the inferred semantic rules against the traces of the buggy version and output rules that are violated in the buggy traces (4). We assume these violated rules are potentially related to the semantic failure. Further optimizations are applied to remove noises and redundancies (5). In the online stage, Oathkeeper only performs minimal instrumentation that is relevant to these final semantic rules from the offline stage. The event tracer ingests traces from the system in real time. The Oathkeeper verifier continuously checks the traces against the deployed semantic rules and reports violations (6).

### 9.2 Instrumentation and Trace Generation

For both inferring semantic rules and runtime verification, we need to first instrument the system to obtain execution traces. The Oathkeeper traces use a uniform event schema that captures operation-related events and state-related events.

Oathkeeper designs a load-time instrumentation library that performs bytecode manipulation when a target system is loaded. This way of instrumentation is convenient (without re-compiling and re-packaging the system) and transparent.

To record operation events, the library adds hooks at the beginning, return and exception point of a method. To record state events, Oathkeeper takes a patch plus base approach. It analyzes the given semantic failure patch and automatically includes the list of classes involved in the patch file. Users can optionally specify names of some important system classes, such as SessionTrackerImpl. With the combined list of classes, Oathkeeper performs simple analysis at the loading phase of these classes to retrieve their member variables of primitive or collections types, and treat them as the state variables. It then identifies instructions that update these variables and insert a hook to emit a state update event with the relevant context (variable name, location, etc.).

For each given test, Oathkeeper switches the target system to the patched version. The tool executes the test with the instrumented system and generates the trace of events. Then Oathkeeper reverts the target system to the buggy version (1). We need to first instrument the system to obtain execution traces. This will generate two sets of traces (2). The verifier continuously checks the traces against the deployed semantic rules and reports violations (3).

### 9.3 Template-Driven Inference

A key challenge in the semantic rule inference step of Oathkeeper is to integrate domain knowledge without requiring significant manual effort, while also having reasonable accuracy and efficiency. We take a template-driven approach to address this challenge. We first summarize general semantic rule patterns, such as happens-before relationship, atomicity, periodicity. For each pattern, we define one or more parameterized templates, such as a state change event for s must happen before the completion event of operation p. Oathkeeper currently supports 18 templates. Table 3 shows some templates integrated in Oathkeeper.

<table>
<thead>
<tr>
<th>Template</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p \rightarrow q$</td>
<td>decommission a datanode should trigger reconstruction</td>
</tr>
<tr>
<td>$s \uparrow \Rightarrow p$</td>
<td>when datanode changes, associated watcher notifies clients</td>
</tr>
<tr>
<td>$s \uparrow \Rightarrow k \uparrow$</td>
<td>after session disconnection, ephemeral node is removed</td>
</tr>
<tr>
<td>$(s = c) \oplus q$</td>
<td>read-only server should not provide write access</td>
</tr>
<tr>
<td>$p + \Delta t \Rightarrow q$</td>
<td>inserted data should expire after the TTL is reached.</td>
</tr>
<tr>
<td>$s \uparrow \Rightarrow q$</td>
<td>cf schema should be altered before alter command returns</td>
</tr>
<tr>
<td>$p \Rightarrow \Diamond (s \uparrow, k \uparrow)$</td>
<td>after snapshot renaming, either new snapshot creation and old snapshot deletion both occur or none of them occur</td>
</tr>
</tbody>
</table>

The trace scale is usually moderate, because it is generated from tests. For example, with ZooKeeper, even under the full instrumentation mode (instrumenting all classes), most end-to-end tests generate less than 10,000 events. A common scale is several thousands. We see large traces in only 5/273 tests that produce over 500,000 events. Under the diff mode (only instrument the classes affected by the patch), the trace typically has hundreds of events.
### Algorithm 1: Generic inference and validation workflow.

**Input:** $L$: a trace (list of events)  
**Output:** a list of inferred invariants (one inv. is a template w/ context)

**Func Infer(L):**  
/* get unique events in the trace (we define equality individually for different types of events) */  
unique-events ← Set(L)  
prescan(unique.events)  
foreach event ∈ L do scan(event)  
return postscan()

**Func Verify(L, context):**  
foreach event ∈ L do  
if scan(event, context) then break  
return postscan()

**Func Main(L_{patched}, L_{buggy}):**  
inv_list ← []  
foreach inv ∈ Infer(L_{patched}) do  
if Verify(L_{buggy}, inv.context) == InvState.FAIL then  
inv_list.add(inv)  
return inv_list

several examples. Our technical report [47] shows the full list.

The inference engine implements an inference algorithm for each template. The algorithm checks if there are matches in a given trace and derives concrete values to each template parameter if so. We call each match a context for the template, which is a potential invariant. For one template (e.g., $p \Rightarrow q$), a trace can have multiple contexts (e.g., $a1 \Rightarrow a2$ and $a1 \Rightarrow a3$).

The templates allow encoding domain-specific semantics without significant specification effort. They also restrict the search space so the inference engine only analyzes trace events that match the template structure and parameter types. While these templates may not represent the exact or full semantics like a high-level specification does, they can capture the essential ingredients for making the semantics hold.

The inference engine takes the trace obtained from running the regression tests against the patched system. Each template class implements an infer function that returns a list of rules from the trace. Most templates follow three phases in the infer function: pre-scan, scan, and post-scan (interfaces defined in Listing 1). The pre-scan step typically builds an index of the unique event set in the trace. The uniqueness is determined by a custom function we define for different types of events. For example, operation invocation events are unique based on the signatures of invoked functions. The scan step iterates through each event in the trace and updates bookkeeping data structures such as an event occurrence map. The post-scan step generates invariants based on the bookkeeping data structures. Templates that do not follow this pattern can customize the procedures. For example, the AfterOpAtomicStateUpdateTemplate iterates forward once and scans backwards once; the StateEqualsDenyOpTemplate scans the trace for each state type in the test.

The core inference algorithm for each template, while different, is relatively straightforward. It essentially involves identifying events in the trace that match the type of a template’s parameter, enumerating hypotheses (candidates) from the contexts, and validating the hypotheses against the trace. Since the trace size is moderate, we can afford enumerations.

**Example.** We describe the inference of a representative template $p \Rightarrow q$, which represents that every invocation of oper-
The scan step goes over the events in the trace and updates the steps are listed in Algorithms 1 and 2.

The steps are listed in Algorithms 1 and 2.

We use Figure 9 (a) to show the process of inferring rules of template \( p \Rightarrow q \) from a patched trace \([e_1, e_2, e_3, e_1, e_2]\). The algorithm assumes all pairs \( <e_i, e_j> \) in the unique event set are candidate contexts to the template, in which \( e_i \) and \( e_j \) are of OpTriggerEvent type and the uniqueness is based on the operation name. Then it attempts to find counterexamples to invalidate wrong rules. The inference algorithm of this template uses a simple counting approach that runs in three steps.

The pre-scan step constructs a nested map \{event: \{event: int\}\} to record the occurrences for the event pairs. For each event pair, the counter is initially zero. Then the scan step iterates through each event \( e \) in the trace in order. If \( e \) is \( e_i \), i.e., a key in the nested map, we increment the counters for all entries with \( <e_i, *> \) keys; if \( e \) is \( e_j \), we decrement the counters for entries that have \( <* , e_j> \) keys and have positive counters. In the post-scan step, we check the final state of counters. If the final counter does not reach zero, there is an orphan \( e_i \) that does not have subsequent \( e_j \). We get \( e_1 \Rightarrow e_2, e_3 \Rightarrow e_1 \) and \( e_3 \Rightarrow e_2 \) at the end. Rules like \( e_1 \Rightarrow e_3 \) are removed because no subsequent \( e_3 \) occurs after the second \( e_1 \).

### 9.4 Rule Validation

After step 8, the inference engine could infer many likely semantic rules. Oathkeeper then applies these rules against the buggy traces (9) and sees which rules are violated. Similarly to inference, each template class needs to implement a verify function. The verify function also usually consists of three phases: pre-scan, scan, and post-scan. The pre-scan step initializes auxiliary data structures specific to the template. The scan step goes over the events in the trace and updates the data structures. In some template, the scan step does not need to iterate through all events in the trace if a contradictory example is already found. The post-scan step checks the data structures and returns the result, which could be PASS (rule is activated and no contradiction is found), INACTIVE (the antecedent of the rule does not occur, e.g., \( p \Rightarrow q \) is inactive in a trace without occurrences of \( p \)), or FAIL (at least one contradiction is found). We only preserve rules that pass in the patched trace and fail in the buggy trace.

#### Example

Algorithms 1 and 2 show the steps to verify template \( p \Rightarrow q \). We use Figure 9 (b) to show the process of validating inferred rules from (a) on a buggy trace \([e_1, e_2, e_3, e_1]\). There are three rules to verify: \( e_1 \Rightarrow e_2, e_3 \Rightarrow e_1, e_3 \Rightarrow e_2 \). In the pre-scan step, we first initialize a counter for each inferred rule. The scan step then updates the counter: for rule \( e_i \Rightarrow e_j \), if a processed event \( e \) matches \( e_i \), we increment the counter; if \( e \) matches \( e_j \), we decrement the counter if it is positive. All three rules are active as both \( e_1 \) and \( e_3 \) appear in the trace. The post-scan step marks rules with non-zero counters as FAIL: \( e_1 \Rightarrow e_2 \) and \( e_3 \Rightarrow e_2 \).

However, there could still be a significant number of rules due to noises like unfinished tests (e.g., an assertion failed in the middle of the test), new type events (new methods introduced), coincidence, and methods that are used for testing only. To reduce these noises, the verifier validates (6) the candidate rules against traces obtained from all test cases, under the patched version, and discards rules that do not hold in all traces. In addition, we filter uninteresting rules about the system start-up or shut-down methods or thread run methods. This is achieved by inserting special marker events at the start and end of test method, and only running the inference algorithms on trace region within the markers.

### 9.5 Runtime Checking

Oathkeeper deploys the refined semantic rules with the target system in production, along with the verifier and event tracer. Oathkeeper performs load-time instrumentation to the production system in a wrapper class of the entry points. Different from the offline stage, the instrumentation is selective to only the deployed rules and is thus lightweight. The event tracer stores in-memory traces from the target systems.

The runtime verifier schedules periodic tasks that validate the current trace against each of the deployed semantic rules. It reuses the same checking logic defined in the function verify of the template. When the engine finds a semantic rule reported as FAIL, it records the counterexamples in the traces for debugging. It also schedules a second check on this violated rule again shortly to tolerate transient violations or inconsistencies in the trace. For high availability, Oathkeeper generates alerts in the log upon detection of potential semantic failures and does not attempt to crash the system.

### 9.6 Optimizations

The validation step can be time-consuming. With \( N \) (often thousands) candidate rules and \( M \) (often hundreds) test cases, we need to get \( M \times N \) traces and check \( N \times M \) times. To reduce the validation time, we introduce a survivor optimization. After a test finishes, we validate the rules, if some rule is already “killed” (invalidated) by this test’s trace, it will not be carried over to the remaining tests. Therefore, only the survived rules will be validated to the end. Another optimization is to run more closely related tests first. The rationale is that some test takes a long time to run but is irrelevant to a given rule (thus the test’s trace will not disprove the rule). By prioritization, we can potentially invalidate false rules faster.
We also add several optimizations to reduce the runtime overhead. First, the event tracer only preserves the most recent events within a time window, since always checking full traces from the start is wasteful. The time window is configured larger than checking frequency to avoid missing checking events. The events involved in time-related semantic rules are excluded as their expiration time is based on their parameters. Second, to achieve both high concurrency and low memory pressure, we decouple the checking from the event emission with a ring buffer design inspired by high-performance message queues [10]. Third, to avoid massive new object creation frequently triggering garbage collection, we reuse expired event objects in the ring buffers. Oathkeeper also pre-allocates buffers for each type of events at the instrumentation phase to prevent buffer initialization blocking.

9.7 Implementation
We implement Oathkeeper in Java (JDK 8). Its instrumentation library is built based on Javassist for class bytecode manipulation. Its test engine leverages JUnit to manage and execute test cases. The tool also includes a workflow script such as checking out patched and buggy versions and checking a semantic rule against given traces.

9.8 Limitations
Our approach makes several assumptions: 1) semantics should be expressible with simple relations of events; 2) the system has a number of test cases with good quality; 3) the failure patch should not involve significant redesign or interface interfaces. If some assumption does not hold, Oathkeeper may fail to deduce good semantic rules.

10 Evaluation
We have integrated Oathkeeper with ZooKeeper, HDFS and Kafka. We evaluate (1) whether Oathkeeper can leverage past semantic failures to check new violations; (2) what runtime overhead it incurs to the target system. The experiments are done in servers with 20-core 2.2 GHz CPUs, 64 GB memory, running Ubuntu 18.04. The Oathkeeper check engine is configured to schedule and check rule violations every second.

10.1 Generation Overview
Oathkeeper requires old semantic failures and their associated regression tests as input to extract semantic rules. We select old semantic failures and their regression tests to reproduce (8 for ZooKeeper, 10 for HDFS and 8 for Kafka). These tests cover major functionalities of the three systems. We add a switch in the system code to easily enable and disable the patch for the semantic failure bugs. We then apply Oathkeeper to the source code to add instrumentation points, run the regression tests with the patch switch turned on and off, and execute other steps in Oathkeeper (Section 9.1). For each case, Oathkeeper infers many raw semantic rules. After

<table>
<thead>
<tr>
<th>JIRA Id</th>
<th>Violated Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZK-1496</td>
<td>ephemeral node should be deleted after session expired</td>
</tr>
<tr>
<td>ZK-1667</td>
<td>watcher should return correct event when client reconnected</td>
</tr>
<tr>
<td>ZK-3546</td>
<td>container node should be deleted after children all removed</td>
</tr>
<tr>
<td>HDFS-14699</td>
<td>failed block need to be reconstructed</td>
</tr>
<tr>
<td>HDFS-14317</td>
<td>edit log rolling should be activated periodically</td>
</tr>
<tr>
<td>HDFS-14633</td>
<td>file rename should respect storageType quota</td>
</tr>
<tr>
<td>KAFKA-12426</td>
<td>partition topic ID should be persisted into metadata file</td>
</tr>
</tbody>
</table>

Table 4: Evaluated newer semantic failures.

the validation and optimization step, the rule set is significantly reduced. In total, Oathkeeper extracted 285 rules for ZooKeeper, 1,209 rules for HDFS, and 150 rules for Kafka.

10.2 Checking Newer Violations
We evaluate whether the inferred rules are useful to catch new semantic failures. Given Oathkeeper’s approach, it is likely less effective with unseen semantics. We reproduce 7 newer (9–34 months later) failures (Table 4) that violate related semantics in the old cases, but with different root causes. With the inferred rules, Oathkeeper detects violations for 6 of them. These newer violations are known bugs by the time we conducted this experiment. However, their root causes and triggering conditions are completely different from the failures used to extract semantic rules. Oathkeeper detects these newer violations with only knowledge from the old failures, which demonstrates the tool’s detection capability.

We show one example in Figure 10. ZK-1496 is not in our study dataset, but its symptom is similar to a studied failure ZK-1208 that was reported 9 months ago prior to ZK-1496 in an older release. Users found that the ephemeral znodes were not deleted long after the client exited. The root cause is a race condition bug that while the session tracker is removing the expired session, another thread is processing an ephemeral node creation request. In ZK-1208, developers added a fix to mark sessions as closing to prevent ephemeral node creation on expiring sessions, and introduced a regression test. Oathkeeper executes the regression test on ZooKeeper twice with patch enabled and disable, and generates two traces (c) and (d). Then Oathkeeper infers rules (e) from the patched traces. Not all inferred rules are useful. Oathkeeper only preserves rules that fail in buggy traces and pass all tests (f). Rules such as (e) are filtered when being validated on all tests. Finally, two verified rules (f) and (g) detect the violations (g).

Oathkeeper fails to detect ZK-1667: client A sets a watch on /d and then disconnects, client B deletes /d and recreates it; when client A reconnects, it receives a NodeCreated event instead of NodeDataDeleted event. The violated semantics fits into one of our templates. However, due to the quality of the old watch test in our pool, Oathkeeper infers other rules.

The average detection time is 0.91 seconds. This result does not contradict with the long-lived semantics finding in Section 7. In the experiments, we trigger the conditions to reproduce the failure soon and measure the detection time from the start time of the violation.

We compare Oathkeeper with a state-of-the-art invariant
detects a median of 5 failures. For example, rules from

Figure 12: Time to generate trace, infer rules, and verify rules against
test suite. ZK: Id 0–7, HDFS: Id 8–17, KF: Id 18–25.

Table 5: System throughput (op/s) with varying percentages of sem-

30,759.49 30,546.00 30,377.50 30,246.04 30,183.15
10
FAIL
20
418.27 417.63 416.71 416.55 416.1
8
174.55 174.56 172.10 172.10 172.06
6
12
Figure 10: Example: Oathkeeper workflow of using ZK-1208 to detect ZK-1496.

Figure 11: Detection of 22 semantic failures in ZooKeeper (sorted by
the bug ticket time in ascending order) when applying Oathkeeper on a sliding subset of the failures for inferring semantic rules.

case Opcode.create:
checkSession();
if (session == null)
for (int i = 0; i < 10; i++) {
// open a connection
ConnectRequest connReq = new ...;
// close connection
RequestHeader h = new ...;
CreateRequest createReq = new ...;
assertEquals(h, zk.getChildren("/").size());
}
found the overhead mainly comes from memory and GC instead of synchronization, which motivated our ring buffer design (Section 9.6) that significantly reduced the overhead.

10.5 Rule Activation and False Positive
We deploy the inferred rules to a cluster of ZooKeeper, HDFS, and Kafka instances. We run a set of workloads against the instances. We first measure the rule activation ratio during the experiment. A rule is activated if the check engine finds the antecedent of the rule has occurred. For ZooKeeper, 11% of the rules are activated. The remaining rules are not activated due to the lack of workloads, faulty conditions, etc., to trigger the antecedent events. For HDFS and Kafka, the activation ratio is 66% and 48%. We then measure the false positive ratio among the activated rules. The result is 4% for ZooKeeper, 9% for HDFS, and 12% for Kafka. This result benefits from the validation steps described in Section 9.4: Oathkeeper eliminates falsely inferred rules by validating the rules against both the buggy trace and the traces from all test cases of a target system. Adding profile runs or a dynamic ban mechanism can further remove the false rules.

11 Related Work

Semantic Bugs. Several studies [42, 55, 58] analyze the prevalence of semantic bugs in open-source server software. Our study analyzes semantic failures in distributed systems. Beyond the difference that we investigate distributed systems, the study of bugs in is in general a complementary effort to the study of failures. The former focuses on analyzing the static code patterns, while the latter focuses on the dynamic manifestations and system misbehavior.

Several solutions are proposed to detect semantic bugs in file systems and DBMS, including cross-checking multiple file system implementations [50], fuzzing [39], and testing using pivoted query [54]. Both cross-checking and fuzzing focus on finding bugs offline. Oathkeeper focuses on a complementary direction of inferring semantic rules for runtime checking. We hope our study can motivate future work to extend these solutions to detect semantic bugs in distributed systems. We observe some open challenges to cross-check distributed systems: distributed systems usually provide a wide variety of semantics that are less rigorously specified compared to file systems, which have well-defined semantics (e.g., POSIX standard) and many implementations. Each distributed system has its unique semantics and may not be cross-checkable. In addition, they often contain many internal and background mechanisms that provide semantic guarantees but the semantics are not easy to be tested. For fuzzing, the challenge is that many silent semantic violations require external faulty events (e.g., node restarts, network error) to trigger besides input. Thus, fault injection testing is needed.

Distributed Systems Failure Study. Understanding failures has been an important theme in distributed system literature, with a series of empirical studies [16, 17, 19, 25, 31, 32, 36, 37, 46, 51], e.g., on fail-slow faults [32], gray failures [37], and network partitions [17]. These failures usually have some error signals such as timeouts. Our study complements these studies and focuses on the under-explored silent semantic failures in distributed systems.

Runtime Verification. Prior works have explored runtime assertions to verify distributed protocols [43, 44], file systems [28], and network functions [57]. Runtime verification [34] is also studied in embedded systems and Java benchmark programs [24]. Recent works [45, 46] propose intrinsic watchdogs that detect partial faults with clear error signals. Lu et al. propose a runtime checker for consistency violations [48]. Overall, there is a lack of runtime verification solutions for monitoring the semantic correctness of large-scale distributed system implementations. Our proposed tool Oathkeeper explores automatically extracting semantic rules to check a variety of semantics for large distributed systems.

Invariant Mining. Inferring likely invariants from software execution traces have been studied, e.g., Daikon [26] and DIDUCE [33]. They mainly focus on mining invariants on the relationship of program variables for single-component software, e.g., off < array.length. These invariants are too low-level to capture the semantics of distributed systems.

Dinv [30] is proposed to infer protocol invariants of program variables across nodes. It runs complex program slicing to instrument program variables influenced by network communication. It then uses Daikon to infer invariants from the logs of running the system’s test suite. I4 [49] infers inductive invariants for verifying distributed protocols.

Oathkeeper is complementary to the two efforts. Instead of protocols and variable relations, we focus on inferring high-level semantic rules for large distributed systems, most of which are not about protocols. Also unlike Dinv, Oathkeeper does not rely on complex static analysis to work and thus does not suffer from analysis inaccuracies and scalability limitations. Oathkeeper takes a unique approach of leveraging past failures and semantic templates to extract semantic rules.

12 Conclusion

Silent semantic violations pose a severe challenge to distributed systems reliability. This paper sheds light on this under-explored yet important problem by presenting a study on real-world failures in popular distributed systems. It reveals that sadly “a promise is often not a promise”. Guided by our study, we design a tool Oathkeeper that automatically extracts semantic rules from past semantic failures, and enforces these rules at runtime to check future violations.

Acknowledgments
We thank our shepherd, Annette Bieniusa, and the OSDI reviewers for their valuable feedback. This work was supported in part by NSF grants CNS-1942794, CNS-2149664, CNS-1910133, and CCF-1918757, and a Facebook research award.
References


[14] ZooKeeper-2774: Ephemeral znode will not be removed when session timeout, if the system time of zookeeper node changes unexpectedly. https://issues.apache.org/jira/browse/ZOOKEEPER-2774.


