CoFI: Consistency-Guided Fault Injection for Cloud Systems

Haicheng Chen
Department of Computer Science and Engineering
The Ohio State University, United States
chen.4800@osu.edu

Wensheng Dou
State Key Lab of Computer Science, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, China
wsdou@otcaix.iscas.ac.cn

Dong Wang
State Key Lab of Computer Science, Institute of Software,
Chinese Academy of Sciences
University of Chinese Academy of Sciences, China
wangdong18@otcaix.iscas.ac.cn

Feng Qin
Department of Computer Science and Engineering
The Ohio State University, United States
qin.34@osu.edu

ABSTRACT

Network partitions are inevitable in large-scale cloud systems. Despite developer's efforts in handling network partitions throughout designing, implementing and testing cloud systems, bugs caused by network partitions, i.e., *partition bugs*, still exist and cause severe failures in production clusters. It is challenging to expose these partition bugs because they often require network partitions to *start* and *stop* at specific timings.

In this paper, we propose *Consistency-Guided Fault Injection* (*CoFI*), a novel technique that systematically injects network partitions to effectively expose partition bugs. We observe that, network partitions can leave cloud systems in inconsistent states, where partition bugs are more likely to occur. Based on this observation, CoFI first infers invariants (i.e., consistent states) among different nodes in a cloud system. Once detecting violations to the inferred invariants (i.e., inconsistent states) while running the cloud system, CoFI injects network partitions to prevent the cloud system from recovering back to consistent states, and thoroughly tests whether the cloud system still proceeds correctly at inconsistent states. We have applied CoFI to three widely-deployed cloud systems, i.e., Cassandra, HDFS, and YARN. CoFI has detected 12 previously-unknown bugs, and four of them have been confirmed by developers.

CCS CONCEPTS

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

KEYWORDS

Cloud system, netwrok partition, fault injection, testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416548

ACM Reference Format:
Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416548

1 INTRODUCTION

Cloud systems are playing an increasingly important role in our daily life. A majority of Fortune 500 companies adopt cloud storage to host their data [6, 11]. Many popular social media websites are backed up by cloud services [15, 27]. As a result, the dependability of cloud systems has become more important than ever. When cloud systems fail, the consequences are usually severe. For example, a three-hour AWS outage in 2017 led to a 150 million dollar loss for S&P 500 companies [10]. As another example, when Facebook service failed in 2014, many Los Angeles residents called 911 [5].

Cloud systems run on networked commodity machines, where network partitions can occur as frequently as once a week and one incident may last for minutes or even hours [2, 17, 19, 40, 41]. Therefore, dependable cloud systems must handle network partitions correctly. Unfortunately, this is a challenging task since network partitions can happen to any node and can *start* and *stop at any time*. Even though developers strive to handle network partitions throughout designing, implementing, and testing a cloud system, network partitions can still lead to cloud system failures [2, 7].

We use a three-node replica-based cloud system in Figure 1 to illustrate the typical process when a cloud system encounters a network partition. Normally, different nodes in a cloud system keep their states consistent by exchanging messages (Phase 1). When a network partition starts, the nodes on the different sides of the partition become disconnected. This may occur when the three nodes have consistent states (Phase 2). However, the two nodes on the left side may change their states (e.g., serving a data update request from the client), causing inconsistency between the two sides (Phase 2 to Phase 3). Similarly, a network partition may occur when the left two nodes are already inconsistent with the right node (Phase 1 to Phase 3). This is possible because in cloud systems node states are updated asynchronously. When a network partition starts, cloud systems often have various built-in mechanisms to recover from inconsistent states, e.g., repeatedly trying to connect the partitioned nodes for recovery. Thus, cloud systems can still

Figure 1: A network partition's effect on a three-node cluster. Each circle represents one node. Circles with the same pattern (i.e., solid, hollow, horizontal stripe, or vertical stripe) represent nodes that are consistent. In Phases 2 and 3, the lightnings represent network partitions, and the dotted arrows represent the message attempts failed by the network partition.

```
1 if (nodeDeadFor1Minute(nodeId)) {
2   status.remove(nodeId);
3 }
```

```
4 if (!status.contains(nodeId)) {
5   status.put(nodeId, nodeStatus);
6 }
```

Figure 2: The triggering process of CASSANDRA-2115. This partition bug can only be triggered when a network partition starts and stops at specific timings.

operate without the partitioned nodes. When the network heals, the two sides of the partition may have inconsistent states (Phase 4). In this case, the cloud system will try to recover the whole cluster back to a consistent state (Phase 5). In this paper, we refer to the cloud system bugs triggered by network partitions as *partition bugs*. A partition bug can occur if the cloud system cannot handle the network disconnection in Phases 2 or 3, or cannot handle the inconsistency caused by the network partition in Phases 3 or 4.

Fault injection is a common technique for testing cloud systems against adversarial conditions, e.g., node crashes and network partitions. Many works inject node crashes when testing cloud systems [1, 29, 30]. However, node crashes and network partitions are fundamentally different. First, node crashes and network partitions often exercise different recovery operations of a cloud system. Second, crashing a node will remove its in-memory state, while partitioning a node will not. So, these node-crashing injection techniques are inapplicable to expose partition bugs in cloud systems. Recently, a few tools have been proposed to inject network failures when testing cloud systems. For example, Namazu randomly drops network packages with a configured probability [32]. However, such a fault model resembles an unreliable network, which is different from network partitions. NEAT [2] and Jepsen [23] can inject network partitions when testing cloud systems, but they rely on developers to specify when a network partition starts and stops.

In this paper, we propose *Consistency-guided Fault Injection* (*CoFI*), an automated technique to expose partition bugs by systematically injecting network partitions into cloud systems. We find that **partition bugs are more likely to occur when cloud systems are running *at inconsistent states*** (i.e., Phases 3 and 4 in Figure 1) due to two main reasons. First, node communications at inconsistent states are harder to reason about than those at consistent states. Second, cloud systems strive to recover system states from inconsistency as quickly as possible, leaving smaller time windows for developers to test cloud systems at inconsistent states. Based on this observation, CoFI's main idea is to inject network partitions to thoroughly exercise cloud systems at inconsistent states.

Specifically, CoFI first employs distributed program invariants [14, 20] to represent the consistent states in a cloud system. A distributed program invariant (or invariant for short) is a property that must hold when multiple nodes are each at certain program point. Using invariants to represent consistent states allows CoFI to automatically identify consistent states in different cloud systems.

Then, CoFI monitors the cloud system's runtime states and starts a network partition when an inconsistent state occurs, i.e., an invariant is temporarily violated. Starting the network partition at inconsistent states can prevent the cloud system from recovering back to consistent states, so that CoFI can thoroughly test the cloud system at inconsistent states. Finally, CoFI systematically explores the stopping point of the network partition based on message types. Hence, CoFI enables more efficient testing of the cloud system than exhaustively stopping the partition for every message.

We have implemented a prototype of CoFI and applied it to multiple versions of three popular cloud systems, namely Cassandra [35], HDFS [39], and YARN [38]. CoFI can successfully detect 4 known bugs, and 12 *unknown* bugs that have never been reported before. Moreover, the triggering processes of 10 out of these 16 bugs have timing requirements on both the starting and the stopping points of their network partitions. At the time of this writing, developers have confirmed four of the 12 unknown bugs reported by CoFI.

In summary, we make the following contributions in this paper.

- We propose *consistency-guided fault injection* (CoFI), a novel approach to expose partition bugs by systematically injecting network partitions at inconsistent states of cloud systems. CoFI can efficiently expose partition bugs that have timing requirements on the starting and stopping points of network partitions.
- We implement a prototype of CoFI[1], and evaluate CoFI using multiple versions of three popular cloud systems, namely Cassandra, HDFS, and YARN. CoFI detects 12 previously-unknown bugs, and four of them have been confirmed by developers.

2 MOTIVATION AND CHALLENGES

In this section, we first motivate consistency-guided fault injection (i.e., CoFI) with a real-world example. Then, we use the motivating example to discuss how CoFI addresses its main challenges.

2.1 A Motivating Example

2.1.1 The Triggering Process. Figure 2 shows the triggering process of CASSANDRA-2115, a partition bug that manifests when two Cassandra nodes exchange gossip messages at a specific inconsistent state. In a Cassandra cluster, each node obtains the status of its peers through exchanging gossip messages. Updates to a node's status are ordered using vector clocks. If C_1, C_2 and C_3 are the three

[1] https://hanseychen.github.io/CoFI/

nodes in a cluster, both C_2 and C_3 will know that C_1 is running normally (Step ①). The cluster is now at a consistent state (Phase 1 in Figure 1). When a user decommissions C_1, C_1 will change its status to "Left", increases its clock value (from t_0 to t_1), and then notifies its peers about the update. When C_2 receives the update message, it will modify its local copy accordingly (Step ②) since the incoming message has a greater clock value ($t_1 > t_0$). However, due to a network partition, C_3 does not receive any message about the update (Step ③). As a result, C_3 will falsely believe that C_1 is still running normally. At this moment, the cluster becomes partitioned and inconsistent (Phase 3 in Figure 1). After certain amount of time, C_2 drops C_1's status, together with the clock value (Step ④). At this point, the network partition heals, allowing C_2 and C_3 to exchange messages at the inconsistent state where C_2 forgets about C_1 while C_3 thinks C_1 is running normally (Phase 4 in Figure 1). We use {∅, Normal} to denote this inconsistent state. C_3 then propagates to C_2 an outdated value of C_1's status (Step ⑤). Since C_2 now knows nothing about C_1, it will blindly accept C_3's value, even though it is outdated ($t_0 < t_1$). As a result, C_1 reappears to be running normally after it has already been decommissioned!

2.1.2 Timing Requirements on Network Partitions. To trigger this bug, C_2 and C_3 need to exchange a gossip message at the inconsistent state {∅, Normal}. This requires the network partition to start after Step ① and before Step ③, as well as to stop after Step ④ and before Step ⑤. First, if the network partition starts before Step ①, C_3 will not consider C_1 as running normally. Second, if the network partition starts after Step ③ or stops before Step ④, C_2 and C_3 will eventually agree that C_1 has left. Finally, if the network partition does not stop before Step ⑤, C_2 and C_3 will not exchange gossip messages at state {∅, Normal}. The bug will not be triggered if any of the aforementioned situations happens. Such a complex timing requirement on the network partition makes the bug difficult to be exposed using random or developer-specified fault injection.

2.2 Challenges and Solutions

To trigger the partition bug in our motivating example, CoFI injects network partitions to thoroughly test the cloud system at inconsistent states. CoFI needs to address the following three challenges.

2.2.1 Challenge 1: How to Represent and Decide Consistent States? The consistency of a system state is closely related to the specific protocols that individual cloud system adopts. For example, Cassandra uses Paxos to replicate user data [36] while HDFS uses replication pipeline [39]. Moreover, even for the same protocol, two cloud systems may have their own implementations such as Cassandra's and Google Spanner's Paxos implementations [18]. CoFI employs *distributed program invariants* [14, 20] to represent the consistent states in a cloud system. A distributed program invariant is a property that must hold when multiple nodes are each at certain program point. For a selected invariant, a cloud system state is consistent if it satisfies the invariant. Otherwise, the state is inconsistent. We can use the following invariant to represent the consistent states in our motivating example, i.e., C_2 and C_3 agree on C_1's status:

$$status[C_1]@C_2 == status[C_1]@C_3$$

From Figure 2 we can see that, at a consistent state, e.g., Step ①, the above invariant is satisfied. Conversely, at an inconsistent state, e.g., Step ②, the above invariant is violated. Since the invariants can be automatically extracted from cloud systems, such consistent states can be easily applied to different protocols and implementations.

2.2.2 Challenge 2: When to Start a Network Partition? An intuitive idea is to start a network partition at every point during the system execution since it simulates the real-world scenario that the network can be partitioned at any time. However, this approach is impractical due to extremely high overhead. In our motivating example, Step ③ alone lasts for more than one minute, containing too many execution points to explore. Instead, CoFI injects a network partition as soon as it detects an inconsistent state at run time. For example, using the invariant in §2.2.1 to represent consistent states, CoFI will start the network partition after C_2 updates its local copy of C_1's status to "Left" (Step ②) and before C_3 updates its local copy of C_1's status to "Left". To provide maximum chances of exposing partition bugs, CoFI injects network partitions at the message level (instead of at the user operation level as employed by other tools [2, 23]) by failing the message exchanges among nodes. Therefore, the gossip messages at Step ③ will be failed, keeping C_2 and C_3 inconsistent.

2.2.3 Challenge 3: When to Stop a Network Partition? To exercise a cloud system at inconsistent states, one can try to stop the network partition at every execution point (i.e., enabling message exchange before each message is sent). However, this approach of exhaustively searching all possible points to stop the network partition has very high overhead. For instance, Step ③ alone consists of more than 100 gossip messages. Trying to stop the network partition before each of them will be inefficient for exposing our motivating bug. To address this issue, CoFI classifies messages into different types and systematically explores the timing of stopping a network partition for each type of messages, instead of each message. After the classification, the gossip messages at Step ③ are grouped into only a few types, drastically reducing the number of stopping points to explore.

3 CONSISTENCY-GUIDED FAULT INJECTION

In this section, we first discuss CoFI's fault model. Then, we explain CoFI's workflow and explain its major steps.

3.1 Fault Model

CoFI tests a cloud system by injecting a period of *temporary* network partition to *one* node in the system. Here, "temporary" means that a started network partition will stop at certain point. Note that starting a network partition at inconsistent states only tests the cloud system at Phase 3 in Figure 1. To thoroughly test a cloud system at inconsistent states, CoFI also stops the network partition to test the cloud system at Phase 4, i.e., exchanging messages among inconsistent nodes.

The specifics of our fault model are as follows. First, in a test run, only one node will be partitioned. Second, the network partition can start and stop at any time (controlled by CoFI), but CoFI will only start and stop the network partition once per test run. Third, during the network partition, all the messages being sent from or

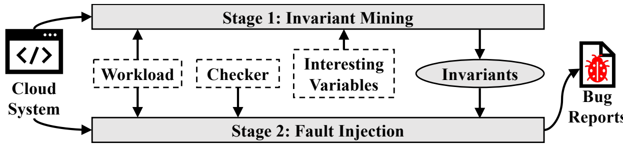

Figure 3: CoFI's workflow. The two solid boxes represent the two stages of CoFI's workflow. The three dotted boxes represent the configurable inputs to each stage.

delivered to the partitioned node will be failed. CoFI focuses on this simple network partition model because it is realistic, and more importantly, cloud systems are expected to correctly handle such a simple fault model at the minimum. It is worth noting that CoFI can be extended to test more complicated fault models, e.g., partitioning multiple nodes and simplex partition [2]. We leave them as future work.

3.2 CoFI in A Nutshell

Figure 3 presents an overview of CoFI's workflow. CoFI works in two stages: *invariant mining* and *fault injection*. In the invariant mining stage, CoFI first runs the cloud system using the workload and records the runtime values of the interesting variables. Then, CoFI mines distributed program invariants from the recorded variable values. The mined invariants will be used to guide starting and stopping network partitions in the fault injection stage. Specifically, for each invariant, which represents consistent system states, CoFI systematically explores the scenarios of network partitions that start at an inconsistent state where the invariant is violated and stop at a later execution point. During each testing run, CoFI uses the checker to detect incorrect system behaviors, e.g., system down. When the checker fails, CoFI will generate a detailed bug report to help developers diagnose the failure. The bug report contains information about the executed workload, the failure symptom, the runtime values of the invariant-related variables, as well as the messages failed by the network partition.

3.3 Specifying Interesting Variables

3.3.1 Which Variables are Interesting? In the invariant mining stage, CoFI mines distributed program invariants based on the runtime values of some interesting variables. Which variables are interesting? We observe that two categories of variables can better represent the state of a cloud system, namely *system metadata* (e.g., the status of a Cassandra node as shown in our motivating example) and *user metadata* (e.g., the location of a container in YARN). This meta-information is critical because anything wrong with this meta-information may seriously affect the reliability of a cloud system. Moreover, an interesting variable should *have data flow from or to the network* so that CoFI can exercise the cloud system in inconsistent states by controlling the timing of the network partition. For instance, the $status[C1]$ variable in our motivating example is a system metadata that has data flow to and from the network. By starting the network partition when C_2 and C_3 are inconsistent on $status[C_1]$ (Step ③ in Figure 2) and stopping the network partition after C_2 removes its $status[C_1]$ (Step ④), CoFI triggers the bug.

```
1 // NameNode's status on NameNode.
2 NameNode.instance.state: NameNode_Status
3 // NameNode's status on DataNode.
4 DataNode.instance.bpManager.bpByNameserviceId.bpServices.state:
        NameNode_Status
```

Figure 4: Two interesting variables in HDFS.

3.3.2 How to Specify Interesting Variables? Since CoFI may access an interesting variable outside of its scope, we represent an interesting variable using the path to access the variable from a Java `static` field (Java's global variable). We refer to these paths as *access paths*. Figure 4 lists two interesting variables (i.e., two access paths) that *refer to the same metadata* in HDFS. Specifically, these two variables both store the status of a NameNode, i.e., whether the NameNode is active or standby. When specifying an interesting variable, one should specify the access path followed by the metadata stored in the variable. Let's use the first interesting variable (Line 2) to further explain how the access path works. `NameNode.instance` is a `static` field that refers to a NameNode object, and `state` is the NameNode object's instance field that stores the NameNode's status. At run time, CoFI accesses this interesting variable by first accessing the `NameNode.instance` object and then accessing the `state` field of that `NameNode.instance` object. Developers can provide their own interesting variables to represent system states, customizing CoFI to test the states they are interested in. The effort needed to specify an interesting variable depends on the implementation details involved in the access path. For example, it is straightforward to derive the first access path in Figure 4 because it matches with "the NameNode's status". Conversely, specifying the second access path requires the knowledge that a DataNode stores information about the NameNodes in the `bpManager` field. The metadata after each access path is a user-defined identifier, which helps CoFI select interesting invariants in the invariant mining stage (§3.4.3).

3.4 Invariant Mining

In the invariant mining stage, CoFI first runs the cloud system and records the interesting variables' values at the program points that likely reflect consistent system states. Then, CoFI groups the values recorded on different nodes to reconstruct the consistent states, from which invariants are mined. Finally, from the mined invariants, CoFI selects the interesting ones to guide fault injection in the next stage.

3.4.1 Which Program Points to Collect Variable Values? To derive consistent states in a cloud system, CoFI mines distributed program invariants from the interesting variables' runtime values at certain program points. The convention is to choose program points like function entrances, function exits, and loop entrances [14]. However, values at these program points may reflect the intermediate results of a node's local computation, which do not represent the system's consistent states. Instead, CoFI selects program points right before a message is sent (**before-send program points**) and right after a message is handled (**after-handle program points**). Specifically, CoFI considers the entrance of a message-sending method as

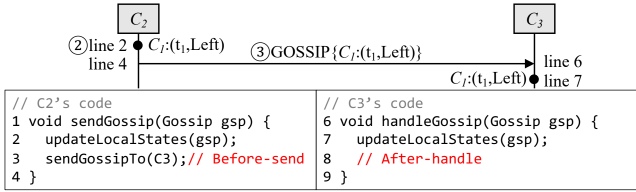

Figure 5: A partial execution of C_2 and C_3 if the network partition in Figure 2 does not occur.

a before-send program point, and considers the exit of a message-handling method as an after-handle program point. A before-send program point reflects the sender node's state when it has applied some changes locally, and is ready to propagate such a change to its peers. Similarly, an after-handle program point reflects the receiver node's state when it has finished updating its local state according to a received message. Therefore, the sender and the receiver of the same message are usually consistent at the pair of these program points.

Figure 5 illustrates such a property using our motivating example. It shows a partial execution of C_2 and C_3 if the network partition does not occur and C_2 sends a gossip message to C_3 at Step ③ in Figure 2. At the before-send program point (Line 3 in Figure 5), C_2 has updated its copy of C_1's status (Line 2). At the after-handle program point (Line 8), C_3 has also updated its copy of C_1's status according to the gossip message from C_2 (Line 7). As a result, C_2 and C_3 are consistent about C_1's status at this pair of program points.

3.4.2 How to Associate Values from Different Nodes? After collecting variable values at the before-send program points and the after-handle program points, CoFI needs to group these values to reconstruct the consistent states. Since the sender and the receiver of a message are usually consistent at the corresponding before-send and after-handle program points, CoFI groups the variable values at the pair of these program points to reconstruct the consistent state. For the example in Figure 5, CoFI groups C_2's variable values at Line 3 and C_3's variable values at Line 8 for the same gossip message to reconstruct the state. The constructed state contains all the variables logged at the two before-send and after-handle program point instances. CoFI considers the state of two nodes, i.e., pair-wise state, instead of the state of all the nodes in the cluster for two reasons. First, many properties of distributed protocols can be captured in pair-wise states. For instance, for Cassandra's gossip protocol, two nodes should be consistent after they have exchanged gossips (e.g., the invariant in §2.2.1). Second, even when a property involves more than two nodes, breaking the sub-property between any pair of the involved nodes will be sufficient to violate the whole property. For example, in HDFS's replication protocol, all the replicas should have the same data when a write succeeds. If any pair of replicas have different data, the whole property no longer holds. When the interesting variable is $status[C_1]$ in Figure 5, grouping C_2's value at Line 3 and C_3's value at Line 8 allows CoFI to mine the desired invariant, i.e., the invariant in §2.2.1.

3.4.3 Which Invariants are More Interesting? After constructing the pair-wise states, CoFI employs Daikon [14] to perform the actual invariant mining. By default, Daikon can mine many invariants [20].

Algorithm 1: Running fault injection tests for an invariant.

Input: invariant, workload, checker
1 iStates := runWithoutPartition(invariant)
2 **while** *iState := iStates.next()* **do**
3   **for** *iNd* $\in$ *iState.inconsistentNodes* **do**
4   passedNewMsgType := true; allMsgTypes := {}
5   **while** *passedNewMsgType* **do**
6   (newIStates, passedNewMsgType) := runWithPartition(iState, allMsgTypes)
7   iStates.add(newIStates)
8 **Function** *runWithPartition(iState, allMsgTps)* **do**
9   workload.start()
10   partition := WAITING; passedNewMsgTp := false
11   **while** *workload.isRunning* **do**
12   **if** *partition = WAITING* $\wedge$ *curState = iState* **then**
13   partition := STARTED
14   **if** *partition = STARTED* $\wedge$ *curMsg* $\notin$ *allMsgTps* **then**
15   allMsgTps.add(curMsg)
16   passedNewMsgTp := true; partition := STOPPED
17   **if** *checker.failed* **then**
18   generateBugReport()
19   **return** (getNewInconsistentStates(), passedNewMsgTp)

Exploring all these invariants can be very time-consuming. Therefore, CoFI further prunes the mined invariants based on a few heuristics, allowing developers to run more interesting tests within a limited budget.

First, CoFI only selects invariants that involve multiple (i.e., two in our setting) nodes. Cross-node invariants capture the consistent states of different nodes. When they are violated, the involved nodes are inconsistent. Second, CoFI removes invariants among variables that refer to different metadata because it may not be meaningful to compare two different metadata. Record that, when specifying an interesting variable, developers also specify the metadata referred to by the variable (§3.3.2). If an invariant involves variables that refer to different metadata, the invariant will be disregarded. By default, Daikon mines many types of invariants, e.g., the equality of variable values (e.g., $var_a == var_b + var_c$), and the membership relation between two variables (e.g., $var_{elmnt} \in var_{set}$). CoFI focuses on equality invariants since it is usually easier to violate these invariants, i.e., to create inconsistent states, during the test runs. Since the invariant for triggering `CASSANDRA-2115` asserts the equality of the same metadata on two nodes, CoFI will select it.

Note that, CoFI's methodology allows using any invariant to represent consistent states, pruning out less interesting invariants improves the test efficiency.

3.5 Fault Injection

In the fault injection stage, CoFI conducts multiple test runs for each mined invariant, as shown in Algorithm 1. More specifically, for each invariant, CoFI first runs the cloud system without injecting network partition to record possible inconsistent states for starting

the network partition in the later runs (Line 1). Then, CoFI iterates over each inconsistent state as the starting point of a network partition (Line 2). For each inconsistent state, CoFI also explores partitioning different inconsistent nodes (Line 3). For each <inconsistent state, partitioned node> pair, CoFI repeatedly runs the cloud system to systematically explore different scenarios of network partitions described as follows.

In each test run for exploring one scenario of a network partition, CoFI first starts the workload (Line 9) and monitors the cloud system's runtime state (represented by the invariant-related variables). Once the system reaches the selected inconsistent state of the current run, CoFI starts the network partition by stopping future messages sending from or delivering to the partitioned node (Lines 12-13). CoFI systematically explores different network partition stopping points by stopping the network partition before different types of messages.

For each message type, CoFI explores two possible cases: stopping the network partition or maintaining the network partition at this point. Specifically, after the network partition starts, CoFI intercepts every message that is sending from or delivering to the partitioned node. If having not seen the type of an intercepted message, CoFI explores the scenario of stopping the partition at this point, which allows the current and future messages to pass (Lines 14-16). Otherwise, CoFI maintains the network partition by dropping the message. CoFI simulates message drop at the application level instead of the OS level. More details will be explained in §4. At the end of each test run, if the checker fails, CoFI generates a bug report for the failure (Lines 17-18). CoFI terminates exploring the scenarios of the network partition for the pair of <inconsistent state, partitioned node> if there is no new message type encountered in the latest run (Line 5), which means CoFI has explored both passing and failing all the message types.

Due to the non-determinism in the cloud system's execution, some inconsistent states may not occur in the first partition-free run but in the later runs. Therefore, CoFI continues collecting new inconsistent states in each test run (Line 19). The newly collected inconsistent states will be used as the starting point of the network partition in the successive runs (Line 7). It is also possible that some inconsistent states CoFI initially collected may not occur in the later test runs. To address this issue, CoFI will retry the test runs multiple times (a configurable parameter) for the inconsistent state.

3.5.1 Identifying Inconsistent States. To identify the possible inconsistent states during the partition-free run (Line 1 in Algorithm 1) and the runs with network partitions (Line 6 in Algorithm 1), CoFI monitors the runtime values of the interesting variables. Specifically, CoFI synchronously collects the variable values at the before-send program points and the after-handle program points. By collecting values at the after-handle program points, CoFI can capture the state change that is caused by a node handling a state-changing message. As shown in our motivating example, after C_2 handles C_1's gossip message at Step ② in Figure 2, CoFI will immediately know that $status[C_1]@C_2$ has become "Left". Sometimes, a state change is not caused by the node handling a message. For example, Step ④ in Figure 2 happens after a timeout (Line 1 in Figure 2). To capture these state changes, CoFI collects the variable values at the before-send program points. Since cloud systems often employ a heartbeat mechanism, collecting variable values at before-send program points helps CoFI periodically refresh its copy of a node's state. When C_2 tries to send a gossip after Step ④ in Figure 2, CoFI will realize that $status[C_1]@C_2$ has become $\varnothing$.

3.5.2 Classifying Messages. When CoFI fails a message during a network partition, the cloud system can react in two ways: The cloud system can simply retry sending the message, or initiate a different protocol to perform recovery (e.g., Cassandra will initiate hinted-handoff when a data replication message fails). Stopping the network partition for retried messages is unnecessary since exchanging the same type of messages will not exercise the cloud system differently. However, stopping the network partition in the second scenario can test if the alternative protocol will proceed correctly at inconsistent states.

Based on this observation, CoFI systematically explores stopping the network partition before each *type* of messages (Lines 14-16 in Algorithm 1), instead of each message. An ideal message type should correlate with the code segments that will be executed when sending or handling the message. In this way, stopping the network partition before different types of messages may exercise different code segments in the cloud system.

CoFI represents the message type using the quad `<stack`, `sender`, `receiver`, `state>`, where `stack` is the runtime call stack of the message sending method, `sender` and `receiver` are the sender and the receiver of the message, and `state` is the system state (i.e., the values of the invariant-related variables) when the message is sent. The message type includes the sender call stack of a message because it reflects the execution path on the sender side. Moreover, messages sent at different call stacks usually belong to different protocols (e.g., Cassandra's gossip and hinted-handoff protocols), or different steps of the same protocol (e.g., the commit-request step and the commit step of a two-phase commit protocol [42]). Therefore, handling these messages will execute different code segments on the receiver side. The other three elements in the message type also correlate with the code segments that will be exercised. In our motivating example, the following three types of messages execute different code segments at the receiver side:

Type 1: <$\text{stack}_{\text{gossip}}$, C_2, C_3, {Left, Normal}>
Type 2: <$\text{stack}_{\text{gossip}}$, C_3, C_2, {Left, Normal}>
Type 3: <$\text{stack}_{\text{gossip}}$, C_3, C_2, {$\varnothing$, Normal}>

Specifically, a Type 1 message will trigger the code that checks the message's vector clock and updates the receiver's state, a Type 2 message will exercise the code that checks the message's vector clock and discards the message, and a Type 3 message will execute the code that blindly accepts the value in the message. For example, all the gossip messages that C_2 sends to C_3 at Step ③ in Figure 2 belong to Type 1. As a result, CoFI will not redundantly try to stop the network partition before each of these equivalent gossip messages.

3.6 Workload and Checker

Workloads drive CoFI to exercise a target cloud system. They can come from various sources ranging from simple unit tests to carefully-crafted test cases for stress testing. Although CoFI can be driven by different workloads, CoFI is most effective when the workload includes cross-node operations that repeatedly read and

write the interesting variables in different ways. With such a workload, CoFI can explore more network partition scenarios. For each of our tested cloud system, we implement a few such workloads using common admin operations (e.g., ResourceManager failover in YARN) and user operations (e.g., file movement in HDFS).

Moreover, developers can flexibly implement checkers to assert the system properties they care about. We provide default checkers in CoFI, one per workload. Specifically, our checkers check for both general failures (i.e., FATAL entries, ERROR entries, and exceptions in execution logs, as well as node crashes) and operation-specific failures (e.g., returning error code and reading stale data).

4 IMPLEMENTATION

CoFI has three components: an instrumentation engine, an invariant mining engine, and a fault injection engine. The instrumentation engine instruments the cloud system to enable reading the interesting variables at run time as well as intercepting message sending and message handling method calls. The invariant mining engine runs the cloud system and mines distributed program invariants from the values recorded by the instrumented code. The fault injection engine runs the workload and the checker on the cloud system and interacts with the instrumented code to inject network partitions.

4.1 Code Instrumentation

We build CoFI's instrumentation engine using Javassist [8], a Java bytecode instrumentation toolkit. To enable accessing interesting variables at run time, the instrumentation engine adds a getter method for each field in the target system. To intercept message sending method calls, the instrumentation engine adds a call to CoFI's `beforeSend()` API at the beginning of each message sending method (i.e., each before-send program point). Similarly, to intercept message handling method calls, the instrumentation engine adds a call to CoFI's `afterHandle()` API at the end of each message handling method (i.e., each after-handle program point). We integrate CoFI with the knowledge of the message sending methods and the message handling methods in popular cloud systems, e.g., `sendOneWay()` in Cassandra. Developers can configure CoFI to instrument different message-passing methods. Inside the message sending methods, the instrumentation engine also adds code to simulate the network partition's effect on the local node. In the fault injection stage, this code will be executed when the fault injection engine decides to fail the message. Developers can also configure the effect of the network partition. By default, the instrumented code throws an `IOException`.

Both `beforeSend()` and `afterHandle()` take three parameters: the sender of the message, the receiver of the message, and the class of the message. All three parameters are used to generate an ID for the message in the invariant mining stage. The sender and the receiver parameters are also sent to the fault injection engine to decide the message type during the fault injection stage.

4.2 Invariant Mining

In the invariant mining stage, `beforeSend()` and `afterHandle()` perform similar tasks: Both APIs first record the interesting variables' values through calling the getter methods. Then, the APIs generate an ID for the message-to-send or the handled message. Finally, they associate the variable values with the message ID and write them to a log, from which the invariant mining engine mines invariants.

Note that CoFI needs to pair the before-send and after-handle program points of the same message to reconstruct a system state. To identify the same message on the sender and receiver sides, CoFI makes two assumptions: (1) Each communication channel between two nodes is FIFO; (2) Messages of the same class go through the same channel. Take our motivating bug as an example, under these two assumptions, the first gossip message that C_2 sends to C_3 is the first gossip message that C_3 receives from C_2. All of our tested systems satisfy these two assumptions. With these assumptions, CoFI constructs the message ID to be the concatenation of the message's sender, receiver, class, and the counter i. In this way, a message will have the same ID on the sender and receiver sides.

When mining invariants, the invariant mining engine first groups the variable values at the before-send program point and the after-handle program point of the same message to form a system state. The engine then concatenates the states of the same program point pair (i.e., same sender, same receiver, and same message class) to form a trace of the states for that program point pair. Afterwards, the engine runs Daikon [14] on the traces to mine invariants. Finally, the mined invariants are pruned based on the rules in §3.4.3.

4.3 Fault Injection

In the fault injection stage, the `beforeSend()` and the `afterHandle()` APIs record the runtime values of the invariant-related variables and report them to the fault injection engine. The `beforeSend()` API also reports the pending message sending event to the fault injection engine, and waits for the engine's decision on whether the message should be failed. If the engine decides to fail the message, the `beforeSend()` API will return a `false`, triggering the execution of the instrumented code in its caller (i.e., the message sending method) to simulate the network partition, e.g., by throwing an `IOException` to signal the caller about the network partition, or by returning from the message sending method to simulate a silent message drop.

5 EVALUATION

Our evaluation aims to answer three research questions: (1) How effective is CoFI in detecting partition bugs in cloud systems? (2) How does CoFI compare with other approaches for injecting network partitions? (3) How efficient is CoFI? We perform our evaluation using a CloudLab [12] machine that runs Ubuntu 16.04. The machine has 40 Xeon® E5-2660 processors and 157 GB memory.

5.1 Experimental Methodology

5.1.1 Target Cloud Systems. We select three widely-used open-source cloud systems as our experiment subjects, i.e., Cassandra [35], HDFS [39], and YARN [38]. They represent different kinds of cloud systems. First, they provide different functionalities: Cassandra is a distributed NoSQL database, HDFS is a distributed file system, and YARN is a distributed computing framework. Moreover, these systems adopt different system architectures: Cassandra is a peer-to-peer system while HDFS and YARN are coordinator/worker systems. Finally, to combat network partitions, these systems implement different recovery mechanisms, e.g., HDFS employs data

Table 1: Experimental settings for the target systems. "Var #" shows the numbers of the interesting variables for each system.

System	Operations in the Workloads	Interesting Metadata	Var #
Cassandra-3.11.5	Create keyspace/column family, read/write data, add/remove column, decommission node	Node status, node token, keyspace name	3
HDFS-3.3.0	Read/write file, move file/directory, failover NameNode, failover DataNode	DataNode ID, NameNode ID, NameNode status, data block ID, data block location	9
HDFS-2.10.0			9
YARN-3.3.0	Launch/stop application, failover NodeManager, failover ResourceManager	NodeManager ID, container ID, container location, ResourceManager ID, ResourceManager status	9
YARN-2.10.0			9

Table 2: The known bugs used to evaluate CoFI. "S" shows whether stopping a network partition is needed to trigger the bug. "Operations" shows the operations in the workload for exposing each bug.

Bug ID	S	Operations	Interesting Metadata
CASSANDRA-3975	✓	Write data, drop table	Column family name
CASSANDRA-2115	✓	Decommission node	Node status
HDFS-14372	×	Shutdown DataNode	DataNode ID
YARN-4288	×	Start cluster	NodeManager ID

re-replication to recover inconsistent user data [39], and Cassandra uses gossip to recover inconsistent system metadata [13].

5.1.2 Detecting Partition Bugs. To evaluate CoFI's effectiveness, we apply CoFI to our target systems and check if CoFI can detect both known bugs and unknown bugs.

Detecting Known Bugs. First, we collect several known partition bugs by inspecting the recently published bug datasets [2, 7, 25, 30]. If a bug satisfies the following requirements, we select it to evaluate CoFI: (1) It happens in our target systems. (2) It only requires partitioning one node to trigger. (3) We can manually reproduce the bug. Finally, we obtained four partition bugs, as shown in Table 2. These four known bugs cover all three target systems, and have different timing requirements on the network partition (Column **S**). Table 2 also shows the operations in each bug's workload and the metadata stored in the interesting variables which we specify for each bug. The operations are extracted from the bug reports. The interesting variables are identified through understanding each bug's triggering process.

Detecting Unknown Bugs. To evaluate CoFI's effectiveness in detecting unknown bugs, we apply CoFI to test the latest versions of our target systems. For HDFS and YARN, we test both their version 2 and version 3 since these versions are both widely deployed and under active development. For Cassandra, we only test its version 3 since the latest minor release of its version 2 (Cassandra-2.2) will no longer be supported after Cassandra's next major release [37]. Table 1 lists the selected system versions.

We design several workloads for each target system using the common user and admin operations as shown in Table 1. For HDFS and YARN, we use the same set of operations for both of their versions. The operations in each workload follow natural order, e.g., create a table before writing data to it. For Cassandra-3, we implement four workloads on a three-node cluster to test regular data access, Paxos data access, schema update, and node decommission. For both HDFS-2 and HDFS-3, we design three workloads to test file system operations, NameNode failover, and DataNode failover. Both the file system operations workload and the NameNode failover workload run on a cluster of two NameNodes and three DataNodes while the DataNode failover workload runs on a cluster with two NameNodes and four DataNodes. For YARN-2, we create three workloads: Run a YARN application in a cluster of one ResourceManager and one NodeManager; ResourceManager failover in a cluster of two ResourceManagers and one NodeManager; NodeManager failover in a cluster of two ResourceManagers and two NodeManagers. For YARN-3, we build two workloads: ResourceManager failover when a YARN application is running in a cluster of two ResourceManagers and one NodeManager; NodeManager failover when a YARN application is running in a cluster of two ResourceManagers and two NodeManagers. Moreover, each YARN cluster also runs on top of an HDFS cluster with one NameNode and one DataNode.

Our checkers check for both general failures (i.e., FATAL entries, ERROR entries, and exceptions in execution logs, as well as node crashes) and operation-specific failures (e.g., returning an error code and reading stale data). To explore more network partition scenarios, we limit at most 100 fault injection runs for each invariant.

Table 1 also shows the metadata stored in the interesting variables that we specify for each target system. For Cassandra-3, we specify the interesting variable that stores keyspace name instead of column family name as for triggering CASSANDRA-3975. This is because keyspace names are accessed more often than column family names (to access a column family, one needs to first access the owner keyspace), potentially exposing more system behaviors when two nodes are inconsistent on a keyspace name. To enable accessing the interesting variables in HDFS and YARN, we add two `static` fields to each version of HDFS and three `static` fields to each version of YARN to refer to the objects of the main components in the system, i.e., NameNode, DataNode, ResourceManager, NodeManager, and ApplicationMaster. In total, this only involves modifying 22 lines of code for all four system versions. The manual efforts for specifying the interesting variables are acceptable: One of our authors specified all the interesting variables and implemented all the modifications in the target systems in a few hours, even if he only has a basic understanding of these systems. For the developers of these systems, specifying interesting variables should take much less time.

5.1.3 Comparing with an Alternative Approach. We compare CoFI with injecting network partitions *randomly*. To be more specific, we repeatedly run each workload for the same time as CoFI spends when testing the target systems. During each test run, we inject a

Table 3: Bugs triggered by CoFI. "Stop" shows whether the network partition needs to stop to trigger the bug. "Status" shows whether the bug is *pending* for developer's confirmation, has been *confirmed* by developers, or has already been *fixed*. "Random" shows whether the bug is triggered by randomly injecting network partitions during our experiment. Note that the four known bugs are from older versions of the target systems and we only apply random injection to the latest versions of these systems.

Bug ID		Failure Symptom	Interesting Metadata	Stop	Status	Random
Known bugs	CASSANDRA-3975	Thread keeps crashing	Column family name	✓	Fixed	N/A
	CASSANDRA-2115	Decommissioned node reappears	Node status	✓	Fixed	N/A
	HDFS-14372	NullPointerException	DataNode ID	×	Fixed	N/A
	YARN-4288	NodeManager aborts	NodeManager ID	×	Fixed	N/A
Unknown bugs	CASSANDRA-15758	Thread crashes	Node status	✓	Pending	✓
	CASSANDRA-15548	A created keyspace can't be found	Node status	×	Confirmed	✓
	CASSANDRA-15546	Data read failure	Node status	×	Pending	×
	CASSANDRA-15437	Decommission failure	Node status	×	Pending	✓
	CASSANDRA-11804†	Data access failure	Node status	×	Confirmed	✓
	HDFS-15367	File metadata inaccessible	NameNode status	✓	Pending	✓
	HDFS-15235	NameNode crashes	NameNode status	✓	Confirmed	×
	YARN-10301	Fail to stop a YARN service	ResourceManager ID	✓	Pending	×
	YARN-10294	Misleading error message	Container's location	✓	Pending	×
	YARN-10288	Invalid application state transition	Container's location	✓	Confirmed	×
	YARN-10232	Invalid application state transition	Container's location	✓	Pending	×
	YARN-10231	Misleading error message	Container ID	✓	Pending	✓

†We did not know CASSANDRA-11804 before CoFI exposed it. This bug is previously reported by others in Cassandra-3.5. But the original bug reporter can no longer trigger it in later versions of Cassandra. We are the first one to report this bug in Cassandra-3.11.5.

network partition randomly. The scenario of the network partition is determined before each test run. First, we randomly select a node to inject network partition. Then, we decide when to start and stop the network partition. The starting point and the stopping point of the network partition are represented using the time offset from the start of a test run. We randomly choose a time offset between 0 and the longest test duration to be the starting point of the network partition, and randomly select a time offset between the starting point and the longest test duration to be the stopping point of the network partition. The longest test duration will be updated when a longer test run occurs. If a test run finishes before the starting point/stopping point is reached, the network partition does not start/stop in that test run.

5.2 Detecting Partition Bugs

5.2.1 Overall Experimental Results. Table 3 lists the partition bugs triggered by CoFI. In this table, we show the detailed information about each bug, including the bug's ID in JIRA (**Bug ID**), the failure symptom of the bug (**Failure Symptom**), the interesting metadata in the invariant that guides CoFI to expose the bug (**Interesting Metadata**), whether the bug requires stopping the network partition to trigger (**Stop**), whether the bug has been confirmed or fixed by developers (**Status**), and whether the bug is also triggered by randomly injecting network partitions during our experiment (**Random**).

As shown in Table 3, CoFI successfully detects all four known bugs using the workloads specified in the bug's JIRA report, demonstrating CoFI's effectiveness in detecting known partition bugs.

Table 3 also shows that, CoFI identifies 12 partition bugs using the interesting variables and the simple workloads (common user and admin operations) that we specified for each system (Table 1). All 12 bugs are previously unknown in the system versions we test. At the time of writing, four of the unknown bugs have been confirmed by developers. The exposed unknown bugs have different symptoms, including severe failures like node crashes and data access failures. Note that these bugs only rely on a small set of interesting metadata, i.e., Cassandra's node status, HDFS's NameNode status, YARN's ResourceManager ID, container location, and container ID.

In Table 3, we can also see that CoFI is effective in exposing partition bugs that requires the network partition to stop at certain points. Specifically, 10 out of 16 bugs can only be triggered by stopping the network partition at certain timing. CoFI exposes these bugs by systematically stopping the network partition for each type of messages. On the contrary, it is challenging to expose these bugs using existing techniques that rely on developers to specify when the network partition starts and stops.

5.2.2 False Positive Analysis. Table 4 shows the detailed statistics of applying CoFI to the latest versions of the target systems. In total, CoFI reports 49 *unique* test failures in these systems (Column **Failures**). 15 of these failures are caused by the unknown bugs listed in Table 3 (Column **Bugs**). The remaining failures are mostly false positives (Column **False Pos.**), while one failure cannot be reproduced for diagnosis (Column **Can't Repr.**).

We further investigate the false positives reported by CoFI. We find that, most (28 out of 33) of these false positives are caused by our checkers asserting for operation success while the operation has to fail. For example, in one of the false positives in Cassandra, a data read with quorum consistency (2 out of 3) fails with a "NoHostAvailable" error. This failure matches with Cassandra's

Table 4: The number of unique test failures in each system. A failure can be a bug, a false positive (False Pos.), or undecided if it cannot be reproduced for diagnosis (Can't Repr.).

System	Failures	Bugs	False Pos.	Can't Repr.
Cassandra-3	10	5	5	0
HDFS-3	7	2†	5	0
HDFS-2	9	2†	7	0
YARN-3	17	5	12	0
YARN-2	6	1‡	4	1
Total	**49**	**15**	**33**	**1**

†The two failures in HDFS-3 and the two failures in HDFS-2 share the same two root causes (HDFS-15367 and HDFS-15235).
‡The failure in YARN-2 shares the same root cause (YARN-10232) with one of the failures in YARN-3.

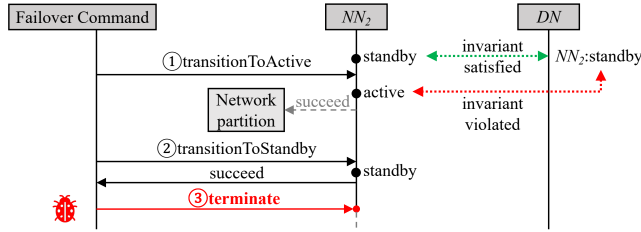

Figure 6: The triggering process of HDFS-15235.

specification because the coordinator node for the read request is partitioned from the other two nodes. As a result, it does not have enough peers (hosts) to serve the request.

These false positives raise a challenge in generating oracles for fault injection tests. Specifically, the correct system behaviors may vary in different fault scenarios. For example, if the "NoHostAvailable" error occurs when only a non-coordinator node is partitioned, the failure is a bug because the coordinator should have enough peers to serve the read request. Automatically generating oracles for fault injection tests will be a challenge for the future research.

5.2.3 Case Study. We now show an example of how partition bugs can occur under intricate network partition scenarios and how CoFI's choice of the partition starting points and the exploration of the stopping points help CoFI expose partition bugs.

Figure 6 shows a partition bug HDFS-15235, which is triggered by a temporary network partition that occurs when a failover attempt is being rolled back. This is a previously-unknown partition bug reported by CoFI. In a cluster with two NameNodes, NN_2 has just became active in a failover attempt (Step ①). Due to a network partition, NN_2 fails to respond to the failover command, which triggers a rollback (Step ②). Normally, as the network partition recovers, NN_2 should safely change back to standby. However, a bug in the rollback process unconditionally terminates NN_2 during the rollback (Step ③). As a result, the cluster loses a healthy NameNode only because of an untimely transient network partition!

CoFI triggers this bug when using the invariant that NN_2 and the DataNode in the cluster (DN) agree on NN_2's status in consistent states. After NN_2 becomes active but before it informs DN, the system is inconsistent. At this point, CoFI starts a network partition on NN_2. During CoFI's systematic exploration of different partition stopping points, it will test stopping the network partition before Step ②, which then triggers the bug. In our experiment, CoFI exposes this bug using only 15 runs when testing HDFS-3. As this example shows, the correctness of a protocol implementation can be hard to analyze under intricate network partition scenarios. CoFI can help expose partition bugs in the implementation by systematically exploring different network partition scenarios. It is also worth noting that to trigger this bug the network partition needs to start and stop *in the middle of* one failover command issued by the admin. Therefore, injecting network partitions at the user operation level, i.e., starting and stopping the network partition before or after the failover command, cannot trigger this bug.

Table 5: The overhead of CoFI.

System	Invariants		Test Runs	
	Mined	Selected	Iterations	Time
Cassandra-3	513	99	9,366	222h, 20m
HDFS-3	157	48	944	24h, 09m
HDFS-2	256	68	1,305	39h, 41m
YARN-3	51	14	1,151	240h, 11m
YARN-2	32	8	250	58h, 59m
Total	1,009	237	13,016	585h, 20m

5.3 Comparing with Random Fault Injection

Table 3 also lists the bugs triggered by randomly injecting network partitions using the policy described in §5.1.3. We find that CoFI is more effective in exposing partition bugs than random injection. Specifically, the random injection only triggerred 6 out of 12 bugs triggered by CoFI. Moreover, the random injection did not trigger any bugs that are new to CoFI. It is also worth noting that if the random injection does not stop the network partition, 3 out of these 6 bugs will not be triggered.

To understand why random injection is not effective in triggering partition bugs, we further analyze the triggering process of four representative bugs (CASSANDRA-15437, HDFS-15367, HDFS-15235, and YARN-10232) to compute the probability to trigger them randomly. First, we assume that the node to partition is correctly selected. Then, based on a *concrete execution*, we compute the probability of selecting the right starting point of the network partition $P(start)$ and the conditional probability of selecting the right stopping point based on the selected starting point $P(end|start)$. Finally, the probability to randomly trigger a bug is computed as $P(bug) = P(start) \times P(end|start)$. We find that the two bugs triggered by both CoFI and the random injection have high likelyhoods to trigger ($P(CA\text{-}15437) = 12.65\%$, $P(HF\text{-}15367) = 8.87\%$), while the other two bugs only triggered by CoFI have much lower probabilities ($P(HF\text{-}15235) = 0.08\%$, $P(YN\text{-}10232) = 0.002\%$). Therefore, both our experiment and our analysis suggest that CoFI is more effective than random injection in triggering partition bugs.

5.4 Overhead Analysis

To measure the overhead of CoFI, we record several metrics while applying CoFI to detect unknown bugs. The metrics include the

number of invariants mined and selected by CoFI, as well as the test iterations and wall clock time CoFI spends in testing each target system. Table 5 shows the values of these metrics.

In the fault injection stage, there are in total 13,016 test runs for all five system versions (Column **Iterations**), which takes about 585 hours to finish (Column **Time**). Specifically, the average time for each test run (Column **Time** / Column **Iterations**) in Cassandra and HDFS is about 1 to 2 minutes. On average, YARN requires more than 10 minutes to finish one test run. This is because YARN employs a wait-and-retry mechanism for many operations. The test time can be shortened by configuring YARN to reduce the wait time and the number of max retries. Given that cloud systems are complicated, the above result demonstrates that CoFI is efficient to be used for real world cloud system testing.

Table 5 also shows, in the invariant mining stage, CoFI mines 1,009 distributed program invariants from the target systems (Column **Mined**). After applying CoFI's invariant pruning strategy, only 237 invariants remain (Column **Selected**). That's said, about 76% of invariants are removed by CoFI's invariants pruning strategy, significantly reducing the number of invariants to test.

6 DISCUSSION

We now discuss limitations and potential threats in our approach.

CoFI's Fault Model. CoFI focuses on a simple fault model: one temporary network partition occurs on one node. Hence, CoFI can miss some partition bugs, e.g., the ones triggered by partitioning multiple nodes. Even though CoFI is not complete, our evaluation shows that CoFI is effective in detecting partition bugs. Extending CoFI to support more fault models can be our future work.

Identifying Interesting Variables. CoFI's effectiveness and efficiency depend on the quality of the specified variables. Specifying uninteresting variables, e.g., variables for local file system data, may prevent CoFI from injecting network partitions at interesting inconsistent states. Moreover, specifying variables that are updated at the same time, e.g., `nodeId` and `nodeId_charArray`, will cause CoFI to test redundant network partition scenarios. To help identify interesting variables, we suggest developers to specify metadata variables that interact with network. Moreover, CoFI provides default interesting variables for our target systems. In the future, it is needed to automate the variable identification process.

Monitoring Variables. CoFI does not use complicated program analysis or expensive synchronization to collect consistent states of the cloud system. Instead, CoFI employs a simple heuristic, i.e., using the before-send and the after-handle program points of the same message to construct system states. This heuristic may cause CoFI to collect an inaccurate state, e.g., if the variables are asynchronously updated in the mean time. In the future, this inaccuracy can be removed by adding synchronization across the cluster.

Threats to Validity. Due to resource limitation, we evaluate CoFI using five versions of three popular cloud systems. Therefore, our experimental results may not reflect the situation in other cloud systems, e.g., distributed streaming systems. However, we strive to be unbiased by selecting systems with different functionalities (i.e., a distributed NoSQL database, a distributed file system, and a distributed computing framework) and architectures (i.e., peer-to-peer vs. coordinator/worker).

7 RELATED WORK

Fault Injection. Fault injection is a commonly used technique for exposing fault-triggered bugs. In recent years, many fault injection techniques have been proposed to expose bugs in cloud systems. NEAT [2] and Jepsen [23] both inject network partitions when testing cloud systems. However, they rely on developers to specify when a network partition starts and stops, which makes them less desirable for exposing partition bugs that have strict timing requirements on network partitions. Moreover, both tools inject network partitions at the user operation level, which limits their effectiveness in exposing intricate partition bugs. On the contrary, CoFI systematically and smartly explores different starting points and stopping points of network partitions at message level, which enables CoFI to expose partition bugs effectively.

Other fault injection techniques include randomly dropping network packets [32], injecting node crashes [1, 29, 30], injecting filesystem faults [16], injecting API failures [3, 4], and reordering network messages [28]. CoFI is complementary to these techniques since it focuses on a different and important fault model for cloud systems, i.e., network partitions.

Fault injection has also been commonly used to test how general software behaves at adversarial scenarios, such as power faults [22, 33, 46] and adversarial inputs [26, 34, 45]. These techniques do not focus on exposing partition bugs in cloud systems.

Bug Detection Techniques for Cloud Systems. Besides fault injection, many other techniques have been proposed to detect bugs in cloud systems. For example, distributed model checkers explore all possible interleavings among network messages and local computation to expose bugs in the cloud system implementations [21, 24, 31, 43]. While being powerful, they still suffer from the state space explosion problem. Some tools can detect bugs by statically analyzing the source code of the cloud systems [7, 9, 44]. Partition bugs involve complex interaction between multiple nodes in the cloud system, which is challenging to analyze statically.

8 CONCLUSION

We present consistency-guided fault injection (CoFI), a novel technique that injects network partitions to expose partition bugs in cloud systems. CoFI is the first fault injection technique that controls both the starting point and the stopping point of the injected network partition. Our evaluation on popular cloud systems shows that CoFI is both effective in exposing partition bugs and efficient to be used in real world cloud system testing.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their thorough and insightful comments. In this work, Haicheng Chen and Feng Qin were partially supported by National Science Foundation grants #CNS-1513120 and #CCF-0953759 (CAREER Award). Wensheng Dou and Dong Wang were partially supported by National Key R&D Program of China (#2017YFB1001804), National Natural Science Foundation of China (#61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences. Both Haicheng Chen and Wensheng Dou are the corresponding authors of this paper.

REFERENCES

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 151–167.

[2] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation*. 51–68.

[3] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Sosenthal, Ali Basiri, and Hochstein Lorin. 2016. Automating failure testing research at Internet scale. In *Proceedings of the 7th ACM Symposium on Cloud Computing*. 1–16.

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

[5] Ben Brody. 2014. LA residents call 911 when Facebook goes down. https://money.cnn.com/2014/08/04/news/companies/facebook-outage-911/index.html.

[6] Box. 2020. Box customers. https://www.box.com/customers.

[7] Haicheng Chen, Wensheng Dou, Yanyan Jiang, and Feng Qin. 2019. Understanding exception-related bugs in large-scale cloud systems. In *Proceedings of the 34th International Conference on Automated Software Engineering*.

[8] Shigeru Chiba. 2020. Javassist. https://www.javassist.org.

[9] Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu, and Peipei Wang. 2018. DScope: Detecting real-world data corruption hang bugs in cloud server systems. In *Proceedings of the 7th ACM Symposium on Cloud Computing*. 333–325.

[10] Datapth.io. 2017. Recent AWS outage and how you could have avoided downtime. https://medium.com/@datapath_io/recent-aws-outage-and-how-you-could-have-avoided-downtime-7d9d9443d776.

[11] Dropbox Business. 2020. Customers - Dropbox business. https://www.dropbox.com/business/customers.

[12] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of CloudLab. In *Proceedings of USENIX Annual Technical Conference*.

[13] Dynamo. 2016. Documentation. http://cassandra.apache.org/doc/latest/architecture/dynamo.html.

[14] Michael D. Ernst. 2000. *Dynamically discovering likely program invariants*. Ph.D. University of Washington Department of Computer Science and Engineering, Seattle, Washington.

[15] Facebook. 2020. Facebook. https://www.facebook.com.

[16] 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 single errors and corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. 149–166.

[17] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM Conference*. 350–361.

[18] Google. 2020. Spanner - Replication. https://cloud.google.com/spanner/docs/replication.

[19] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. 2016. Evolve or die: High-availability design principles drawn from Googles network infrastructure. In *Proceedings of the ACM SIGCOMM Conference*. 58–72.

[20] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. 2018. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*. 1149–1159.

[21] 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 *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*.

[22] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. Crash consistency validation made easy. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 133–143.

[23] Kyle Kingsbury. 2018. Distributed systems safety research. Retrieved Oct 10, 2018 from https://jepsen.io/

[24] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. 399–414.

[25] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. 2016. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*.

[26] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 254–265.

[27] Linkedin. 2020. Linkedin. https://www.linkedin.com.

[28] 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. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*.

[29] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*.

[30] 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.

[31] 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 14th EuroSys Conference*. 20.

[32] osrg. 2018. Namazu: Programmable fuzzy scheduler for testing distributed systems. https://github.com/osrg/namazu.

[33] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. 433–448.

[34] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, Vol. 16. 1–16.

[35] The Apache Software Foundation. 2016. Cassandra. http://cassandra.apache.org.

[36] The Apache Software Foundation. 2016. Cassandra - Data manipulation. https://cassandra.apache.org/doc/latest/cql/dml.html.

[37] The Apache Software Foundation. 2016. Cassandra's older supported releases. https://cassandra.apache.org/download/.

[38] The Apache Software Foundation. 2019. Apache Hadoop YARN. https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[39] The Apache Software Foundation. 2019. HDFS Architecture. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[40] Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, Alex C Snoeren, Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, and Alex C Snoeren. 2012. On failure in managed enterprise networks. *HP Labs HPL-2012-101* (2012).

[41] Daniel Turner, Kirill Levchenko, Alex C Snoeren, and Stefan Savage. 2010. California fault lines: Understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM Conference*. 315–326.

[42] Wikipedia. 2020. Two-phase commit protocol. https://en.wikipedia.org/wiki/Two-phase_commit_protocol.

[43] 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 *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*.

[44] 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 *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. 249–265.

[45] Michal Zalewski. 2017. American fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[46] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. 449–464.