Coverage Guided Fault Injection for Cloud Systems

Yu Gao*†, Wensheng Dou*†‡§, Dong Wang*†, Wenhan Feng*†, Jun Wei*†‡§, Hua Zhong*†, Tao Huang*†
*State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
†University of Chinese Academy of Sciences
‡University of Chinese Academy of Sciences Nanjing College
§Nanjing Institute of Software Technology
{gaoyu15, wsdou, wangdong18, fengwenhan21, wj, zhonghua, tao}@otcaix.iscas.ac.cn

***Abstract*—To support high reliability and availability, modern cloud systems are designed to be resilient to node crashes and reboots. That is, a cloud system should gracefully recover from node crashes/reboots and continue to function. However, node crashes/reboots that occur under special timing can trigger crash recovery bugs that lie in incorrect crash recovery protocols and their implementations. To ensure that a cloud system is free from crash recovery bugs, some fault injection approaches have been proposed to test whether a cloud system can correctly recover from various crash scenarios. These approaches are not effective in exploring the huge crash scenario space without developers' knowledge.**

In this paper, we propose *CrashFuzz*, a fault injection testing approach that can effectively test crash recovery behaviors and reveal crash recovery bugs in cloud systems. CrashFuzz mutates the combinations of possible node crashes and reboots according to runtime feedbacks, and prioritizes the combinations that are prone to increase code coverage and trigger crash recovery bugs for smart exploration. We have implemented CrashFuzz and evaluated it on three popular open-source cloud systems, i.e., ZooKeeper, HDFS and HBase. CrashFuzz has detected 4 unknown bugs and 1 known bug. Compared with other fault injection approaches, CrashFuzz can detect more crash recovery bugs and achieve higher code coverage.

***Index Terms*—cloud system, crash recovery bug, fault injection, bug detection, fuzzing**

I. Introduction

Cloud systems [1]–[7] have become the backbones of modern Internet applications. To achieve high reliability and availability, cloud systems should correctly recover from node crashes and reboots and continue to function. Therefore, complex crash recovery protocols and implementations are introduced to tolerate node crashes and reboots, and play an important role in cloud systems.

However, crash recovery bugs that lie in incorrect crash recovery protocols and implementations pose key challenges for the reliability of cloud systems [8], [9]. Cloud systems usually involve complex protocols and implementations, and each node in a cloud system may crash or reboot at any time. Therefore, the crash scenarios (i.e., the combinations of node crashes and reboots) in a cloud system can be very huge. Node crashes and reboots that occur under special timing can trigger crash recovery bugs, and make cloud systems fail to recover. It is challenging for developers to anticipate all crash scenarios, and to expose crash recovery bugs through in-house testing.

Wensheng Dou and Tao Huang are the corresponding authors.

By injecting crashes and reboots into cloud systems, fault injection approaches can test the protocols and implementations in cloud systems, especially crash recovery behaviors. Random fault injection frameworks [10], [11] inject crashes randomly, but they are difficult to hit corner-case crash recovery bugs, in which crashes should occur under specific timing. Exhaustive fault injection approaches, e.g., FATE [12], combine brute-force search with heuristics to explore the combinations of multiple node crashes. It is not effective in exploring the huge space of crash scenarios. Some fault injection tools further allow developers to express their own fault injection strategies [13], [14]. Similar to exhaustive fault injection approaches, distributed system model checkers [15]–[17] enumerate the orders of non-deterministic events (including node crashes), and suffer from the state space explosion problem. Some approaches, e.g., FCatch [18] and CrashTuner [19] focus on specific crash scenarios. Although existing fault injection approaches have achieved great progresses, they still struggle at searching through the huge state space of cloud systems. Therefore, some of them can only test limited crash scenarios [18], [19], or cannot support node reboots [11]–[13], [15].

In this paper, we propose a novel coverage guided fault injection approach for cloud systems *CrashFuzz*, which can effectively test crash scenarios in a cloud system, i.e., possible combinations of node crashes and reboots, and reveal crash recovery bugs. Compared with exhaustive fault injection approaches, CrashFuzz can smartly generate unique and suspicious crash scenarios. To be specific, CrashFuzz takes I/O points in cloud systems as potential fault injection points, and treats crash scenarios occurring on I/O points as the special inputs of cloud systems. Guided by the system runtime feedbacks, e.g., coverage and I/O information, CrashFuzz adjusts the injection of node crashes and reboots, and prioritizes crash scenarios that can trigger new crash recovery behaviors. Thus, CrashFuzz can potentially reveal new crash recovery bugs.

We have implemented CrashFuzz and evaluated it on three popular open-source cloud systems, i.e., HDFS [20], HBase [21] and ZooKeeper [22]. CrashFuzz has detected 4 unknown bugs and 1 known bug in these systems. Compared with alternative fault injection approaches, e.g., random and brute-force fault injection approaches, CrashFuzz can expose more crash recovery bugs and achieve higher code coverage. We have made CrashFuzz publicly available at https://github.com/tcse-iscas/crashfuzz.

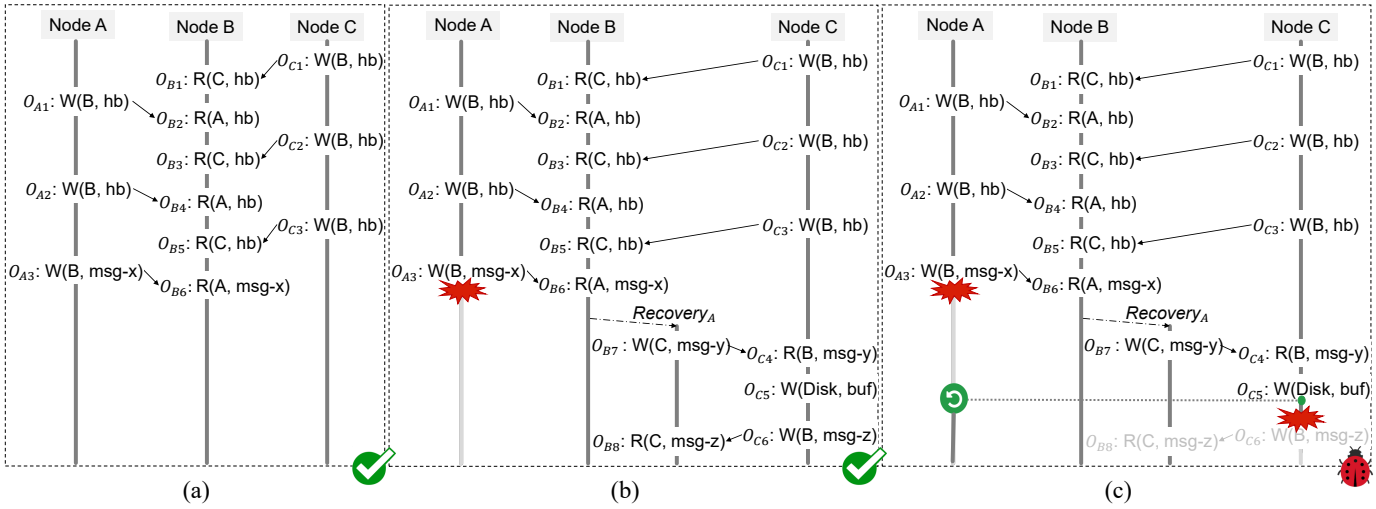

Fig. 1. An illustrative example. W denotes write, R denotes read, hb denotes heartbeat, $\rightarrow$ denotes message sending, O_{A1} denotes the first I/O operation on node A, $W(B, hb)$ denotes writing a heartbeat message to node B, and $R(A, hb)$ denotes reading a heartbeat message from node A.

In summary, we make the following contributions.

- We propose CrashFuzz, a novel coverage guided fault injection approach to expose crash recovery bugs in cloud systems. Based on runtime feedbacks of cloud systems, CrashFuzz prioritizes to test the combinations of node crashes and reboots that are prone to cover new crash recovery behaviors and trigger crash recovery bugs.
- We implement CrashFuzz, and evaluate it on the newest stable versions of three popular cloud systems, i.e., ZooKeeper, HBase and HDFS. CrashFuzz has detected 4 previously-unknown bugs and 1 known bug in them.

II. Motivation

In this section, we use an illustrative example to explain the need for smart fault injection testing for cloud systems, and our solutions.

A. An Illustrative Example

Real-world cloud systems are usually very complex. It is challenging to explain crash recovery by utilizing a real-world cloud system. Therefore, we construct some abstract crash scenarios in Fig. 1 as our illustrative example to show the crash recovery procedure and how a crash recovery bug occurs.

In Fig. 1(a), three nodes, i.e., node A, B, and C, form a master-slave cloud system. Here, node B is the master node, and node A and C are two slave nodes. Node A and C maintain connection with node B through heartbeat messages, e.g., O_{A1}, O_{B2}, O_{C1} and O_{B1}. When node A submits a task through a message (e.g., O_{A3}), mater node B reads the related data from the message (i.e., O_{B6}) and further processes the data.

Nodes in a cloud system may crash or reboot, and trigger crash recovery procedures. Fig. 1(b) shows a crash scenario and its recovery procedure. In this scenario, node A crashes after O_{A3}, and master node B notifies slave node C to take over node A's task and update corresponding data (This recovery procedure is similar to region reassignment in HBase, in which the master node reassigns the data regions on a dead slave node to another slave node). Thus, the recovery procedure of node A results in communication between node B and C (i.e., O_{B7}, O_{B8}, O_{C4} and O_{C6}), and a disk update operation O_{C5}. Note tha, this crash scenario in Fig. 1(b) can be correctly recovered by the system, and cannot result in a crash recovery bug.

In cloud systems, node crashes and reboots can result in specific crash states (i.e., the states of a cloud system after a crash/reboot), which should be correctly recovered to normal states by cloud systems. If a cloud system cannot correctly recover from such crash states, crash recovery bugs manifest [8]. Fig. 1(c) shows such a crash scenario, which can trigger a crash recovery bug. In this scenario, node A crashes after O_{A3}, and then node A reboots after O_{C5}, while node C crashes before O_{C6}. This crash scenario cannot be correctly recovered by the system, and triggers a crash recovery bug.

Note that, node crashes/reboots occurring on other system states in our illustrative example can be correctly recovered by the system, and cannot trigger crash recovery bugs. For example, if node A crashes before O_{A2} (a heartbeat operation), node B only removes node A from its alive node list (e.g., the leader node in ZooKeeper removes a dead follower node from its follower list), and the operations O_{B7}, O_{B8}, O_{C4}, O_{C5} and O_{C6} will not appear since different recovery behavior is triggered. In Fig. 1(c), if node C crashes after O_{C6}, the system can still correctly recover from this new crash scenario, and will not trigger a crash recovery bug.

B. Crash Scenario Space in Cloud Systems

In cloud systems, any node can crash or reboot at any time. However, crashes/reboots on I/O points can create critical crash scenarios. An I/O operation in cloud systems can produce persistent states (e.g., data in files) or store states into other nodes (e.g., send messages). Computation operations

(e.g., $a+1$) can only produce in-memory states. When a node crashes, all in-memory states caused by computation operations disappear, while the impacts caused by I/O operations are left in the cloud system. Therefore, node crashes that occur on different I/O points can produce different crash states, while node crashes that happen between two consecutive I/O operations on the same node produce the same crash state. Therefore, similar to existing approaches [12], [13], we only focus on I/O points for fault injection in cloud systems. In Fig. 1, we only show I/O operations, and ignore other computation operations.

A crash scenario contains one or more node crashes and reboots that occur on I/O points of a cloud system. Such combination of crashes and reboots can cause huge crash scenario space for cloud systems. Take the three-node cloud system in Fig. 1(a) as an example. Without injecting any faults, there are 12 I/O operations in the system (three operations on node A, six operations on node B, and three operations on node C). We can produce (3 + 6 + 3) = 12 crash scenarios by injecting only one crash in the system, and (3*6 + 3*3 + 6*3) = 45 crash scenarios by injecting two crashes on two different nodes. In our experiment, we run HDFS [20] on a 5-node cluster. Without injecting any faults, running a workload for HDFS can produce around 400 I/O operations for each node. We can produce 2,000 crash scenarios by injecting only one crash, about 400^2 * C_5^2 = 1,600,000 crash scenarios by injecting two crashes on two different nodes, and about 400^3 * C_5^3 = 640,000,000 crash scenarios by injecting three crashes on three different nodes. If we further consider injecting reboots, the possible crash scenarios could increase quickly. Therefore, testing cloud systems by enumerating all possible crash scenarios is time-consuming, and may be impossible.

We observe that some crash scenarios may result in similar crash states, and cloud systems take these states equally and use the same recovery behaviors to handle them. For example, in Fig. 1(a), a crash after O_{A1} and a crash after O_{A2} (these two I/O operations send heartbeat messages, and do not change system states) can cause similar crash states and trigger the same recovery behavior, e.g., node B removes the dead node A from its alive node list. This observation inspires us to propose CrashFuzz for smart fault injection testing in cloud systems.

C. Challenges and Solutions

To tackle the huge crash scenario space in cloud systems, and effectively test crash recovery behaviors, we propose a novel fault injection approach CrashFuzz, which can smartly explore various crash scenarios, i.e., the combinations of multiple node crashes and reboots, and reveal crash recovery bugs faster.

Similar to program inputs, injecting node crashes/reboots can also affect the execution of a cloud system. Therefore, a crash scenario can be treated as a special input for cloud systems. Inspired by fuzz testing, i.e., adjusting program inputs according to the feedbacks from program execution, we intentionally mutate the combination of node crashes and reboots according to the system runtime feedbacks, e.g., I/O and coverage information, to guide a cloud system to cover different crash recovery behaviors and increase the chance of triggering crash recovery bugs.

To apply our approach on real-world cloud systems, we need to address the following two challenges.

First, how to generate valid crash scenarios, i.e., the combinations of node crashes and reboots? Given all the I/O points executed in a fault-free cloud system run, we cannot simply enumerate node crashes and reboots on all the I/O points. Some combinations of node crashes and reboots may be invalid, e.g., crashing a dead node. In addition, we aim to test whether a cloud system can tolerate partial node crashes that cannot fail the whole system and are expected to be correctly recovered. Therefore, we do not test crash scenarios that fail the whole system, e.g., crashing all the nodes in Fig. 1(a). To generate such valid crash scenarios, we generate each new crash scenario by injecting only one fault (a node crash or reboot) to an existing crash scenario. The newly injected fault should satisfy some constraints, e.g., only alive nodes can crash.

Second, how to identify and prioritize suspicious crash scenarios? A cloud system may have a tremendous number of crash scenarios, making it impractical to enumerate all of them. To address this challenge, we generate new crash scenarios from existing ones, and design special selection strategies to prioritize suspicious crash scenarios, which are prone to cover new crash recovery behaviors and trigger crash recovery bugs. We prefer to test crashes on unique I/O points. For example, if the crash on O_{A1} in Fig. 1(a) has been tested, we prefer to test the crash on O_{A3} next rather than crashes on O_{A2}, O_{C1}, O_{C2} and O_{C3}. We also prioritize the crash scenarios that inject faults during recovery procedures and contain multiple faults. For example, the crash scenario shown in Fig. 1(c) can be tested earlier. With our prioritization strategies, we can identify and prompt suspicious crash scenarios to perform a more effective fault injection testing.

III. Fault Model

In this section, we describe CrashFuzz's fault model.

A. Fault Point

As discussed in Section II-B, we focus on I/O points for fault injection in cloud systems.

A fault point (we also refer to it as an I/O point in this paper) can be expressed by following information: (1) Node ID, which refers to the node that performs the I/O operation. (2) Call stack for the I/O operation. (3) Source, which refers to the node that sends a message or the file path for a disk read operation. (4) Destination, which refers to the node that receives a message or the file path for a disk write operation. (5) Timestamp. (6) Event type. In our fault model, three types of events can happen on an I/O point: a node crash event ($Crash_{node}$), a node reboot event ($Reboot_{node}$) and a fault-free event. Here, $node$ refers to the node affected by the fault.

B. Fault Sequence

We use fault sequences to represent various crash scenarios. A fault sequence indicates when and where to inject a node crash or reboot to the cloud system under test.

We sort all the fault points executed in a system run according to their timestamp and obtain a fault sequence. A fault sequence can be expressed as follows:

$$FaultSeq = [FtPt^1_{n_x}, FtPt^2_{n_y}, ..., FtPt^n_{n_z}]$$

A fault point $FtPt^i_{node}$ in the fault sequence indicates that the fault point is executed by node $node$, and it is the ith fault point in the sequence.

By default, the event type of every fault point in a fault sequence is the fault-free event, which refers to a fault-free scenario. For a crash scenario that contains one or more node crashes and reboots, we set the event type of specific fault points in a fault sequence to node crash or reboot.

A fault sequence should satisfy the following constraints:

- Only alive nodes can crash. When an alive node crashes, it stops immediately and becomes a dead node.
- Only dead nodes can be rebooted.
- A fault sequence must meet system-specific constraints, i.e., the number of dead nodes should not exceed the maximum number of dead nodes that the target system can tolerate. For example, a fault sequence cannot make all the nodes in the target system crash at the same time.
- A fault sequence must meet user-specified constraints, i.e., the number of total faults in a fault sequence cannot exceed the maximum number of faults specified by developers.

The number of faults contained in a fault sequence can be infinite. For example, we can crash and then reboot a node in the cloud system repeatedly. By using a fault sequence, a crash scenario can be represented as the combination of several node crashes and reboots that occur on specific I/O points.

IV. Approach

Fig. 2 shows the overview of CrashFuzz. CrashFuzz takes a target cloud system, a workload and corresponding constraints as inputs, and outputs the detected crash recovery bugs. CrashFuzz takes several steps to force a cloud system to handle various fault sequences for testing crash recovery behaviors and detecting crash recovery bugs. The possible fault sequences to be tested are stored in the fault sequence queue.

Initial run. At beginning, the fault sequence queue is empty, and we do not have any I/O points that can be used to generate fault sequences. Therefore, in the first step, we initialize the fault sequence queue and perform the initial run without injecting any fault into the cloud system. During the initial run, we collect runtime information, e.g., coverage and I/O information, by instrumenting the cloud system at run time.

Fault sequence generation and mutation (Section IV-A). After a system run, we generate and mutate fault sequences based on the collected runtime information, and add the newly generated fault sequences to the fault sequence queue.

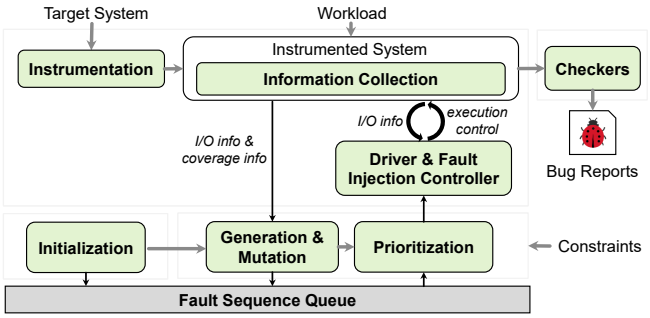

Fig. 2. CrashFuzz overview.

Prioritizing suspicious fault sequences (Section IV-B). After applying a series of prioritization strategies, we obtain a group of candidate fault sequences. We compute a score for each candidate fault sequence. Based on the scores, we randomly select a fault sequence from the candidates to test. The selected fault sequence has higher possibility to cover new crash recovery behaviors and trigger crash recovery bugs.

Fault injection testing (Section IV-C). For a selected fault sequence, we run the cloud system and workload again. Meanwhile, we inject faults to the target system according to the fault sequence under test.

Detecting crash recovery bugs (Section IV-D). During and after the fault injection testing, we use the predefined checkers to check failure symptoms, e.g., system hangs and job/system failures, to detect crash recovery bugs. Bug reports are generated for the tests that do not pass the checkers.

After a fault injection test, we can generate and mutate new fault sequences based on the testing results and collected runtime information again. When the user-specified test time is reached, or there is no fault sequence in the queue waiting for testing, the whole testing terminates.

A. Generating and Mutating Fault Sequences

After the initial run that does not inject any faults into the cloud system, we can collect all the I/O points executed in the run. Then, we can obtain an initial fault sequence without any faults, and perform the initial mutation on the initial fault sequence. For each following test, we generate a new fault sequence $seedSeq$ that reflects the actual execution behaviors first, and then perform the normal mutation on $seedSeq$. After initial mutation and normal mutation, we get a group of new valid fault sequences and add them to the fault sequence queue.

Initial mutation. The initial fault sequence contains all the I/O points executed in the initial system run. All the I/O points in the initial fault sequence correspond to the fault-free events. In the initial mutation, CrashFuzz generates a group of new fault sequences by injecting a node crash event to each I/O point in the initial fault sequence. For example, after initial mutation, four mutated fault sequences can be generated for an initial fault sequence that contains four I/O points.

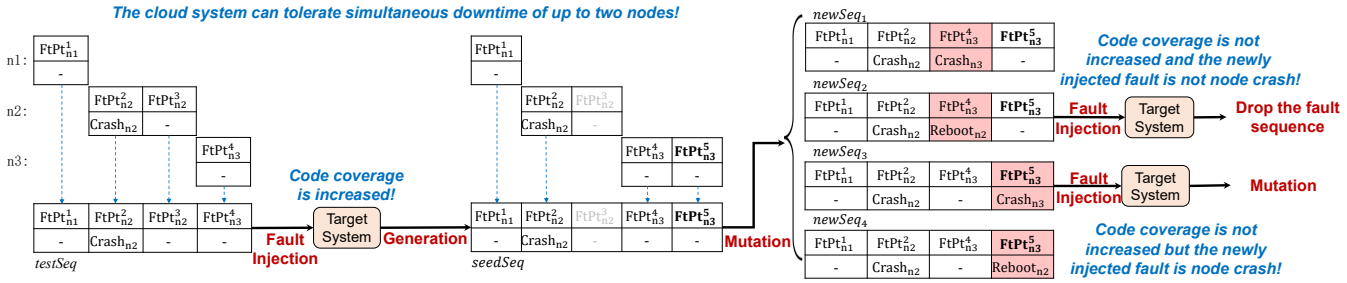

Fig. 3. An example of normal mutation.

Normal mutation. As shown in Fig. 3, for a fault sequence $testSeq$ under test, after successfully injecting all the node crashes and reboots according to $testSeq$, CrashFuzz first decides whether to explore it further. If the test does not cover more code and the newly injected fault (i.e., the last fault in $testSeq$) is not a node crash, CrashFuzz discards $testSeq$, e.g., $newSeq_2$ in Fig. 3. Otherwise, CrashFuzz further performs normal mutations. Here, CrashFuzz still keeps a fault sequence that does not increase code coverage but the newly injected fault is node crash, e.g., $newSeq_3$ in Fig. 3. This is because the system states caused by a node crash can still affect the subsequent reboot behaviors. For example, the dirty local file caused by the node crash can affect the restarted node. The recovery procedure triggered by the subsequent node reboot may increase the code coverage. We keep such fault sequences in case we omit some valuable crash scenarios.

For a fault sequence $testSeq$ that needs further exploration, CrashFuzz first generates a new fault sequence $seedSeq$ based on the collected I/O points and fault events that occur during the test. After a test, some I/O points in $testSeq$ may disappear, and new I/O points may appear. For example, as shown in Fig. 3, after testing, the original fault point $FtPt^3_{n2}$ disappears due to the node crash $Crash_{n2}$, and the new fault point $FtPt^5_{n3}$ appears. Therefore, we generate a new fault sequence $seedSeq$ that can reflect actual execution behaviors.

As shown in Algorithm 1, CrashFuzz mutates $seedSeq$ by adding only a node crash or reboot on $seedSeq$ to generate a group of new fault sequences $newSeqs$. Specifically, CrashFuzz searches from the I/O point right after the last fault in $seedSeq$ (Line 3). For each fault-free I/O point, CrashFuzz tries to mutate $seedSeq$ and generate a new fault sequence by crashing the node that executes the I/O point or rebooting a previously dead node. The newly generated fault sequences will be added to $newSeqs$ (Line 5-14). When injecting a node crash/reboot that affects node $tarNode$ on the ith fault point of $seedSeq$, CrashFuzz first generates a new fault sequence seq that is the same as $seedSeq$, and then sets the event of ith fault point in seq as $Crash_{tarNode}$ or $Reboot_{tarNode}$ (Line 19-22). The new fault sequence seq should satisfy the constraints defined in Section III-B (Line 31-42). Therefore, we can generate valid fault sequences.

Fig. 3 shows an example of normal mutation. In Fig. 3, the distributed system can tolerate at most two node crashes at the same time. The seed fault sequence $seedSeq$ contains a node crash fault, i.e., crashing node $n2$ on $FtPt^2_{n2}$. CrashFuzz explores every fault point after $FtPt^2_{n2}$, and finally generates four new fault sequences. The newly injected faults in these four fault sequences either crash another node except for $n2$ in the system, or reboot node $n2$.

B. Favoring Suspicious Fault Sequences

To explore fault sequences in the fault sequence queue smartly, we first apply a series of prioritization strategies to get a group of suspicious fault sequences in the queue as candidates. Then, we compute priority scores for every candidate fault sequence, and randomly pick a fault sequence from the candidates to test based on their priority scores.

1) Prioritizing Suspicious Candidates: According to Section IV-A, for a fault sequence $newSeq_i$ mutated from $seedSeq$, if it contains $n+1$ faults, then the first n faults have been tested (i.e., $seedSeq$). Therefore, the newly injected fault (i.e., the last fault) in $newSeq_i$ is the critical fault that can potentially trigger new recovery behaviors and crash recovery bugs. When selecting a fault sequence from the queue to test, we mainly focus on the last fault in a fault sequence, and takes three types of fault sequences as suspicious candidates.

First, for all the fault sequences in the queue, we preferentially test the sequences that inject node crashes or reboots on new I/O points. For a fault occurring on an I/O point, we can abstract it as a fault ID. A fault ID is the hash of the call stack information of the I/O point and the fault type (i.e., node crash or reboot). The faults that have the same fault ID are prone to trigger similar crash recovery behaviors, e.g., crashes happen on o_{C1}, o_{C2} and o_{C3} shown in Fig. 1(a).

Second, we preferentially test fault sequences that contain node crashes or reboots occurring during the recovery process. The recovery process that is responsible for handling a node crash/reboot can also crash. Node reboots that occur during the recovery process can cause concurrent recovery behaviors [8]. Such crash scenarios can complicate the recovery behaviors and have not been given sufficient attention in fault injection testing for cloud systems.

CrashFuzz identifies a fault occurring during the recovery process by identifying whether the I/O point where the fault occurs was executed during the recovery process. To be specific, for a fault sequence $testSeq$ under test, we can

Algorithm 1: Mutate fault sequences.

```
1  Function getNewSequences(seedSeq):
2      newSeqs ← ∅;
3      i ← seedSeq.getIndexOfLastFault() + 1;
4      for i < seedSeq.size() do
5          if seedSeq.get(i).event.type ≠ NULL then
6              continue;
7          end
8          n ← seedSeq.get(i).node;
9          seq ← mutate(seedSeq, i, CRASH, n);
10         newSeqs.AddIfNotNull(seq);
11         foreach n ∈ seedSeq.deadNodes do
12             seq ← mutate(seedSeq, i, REBOOT, n);
13             newSeqs.AddIfNotNull(seq);
14         end
15         i ← i + 1;
16     end
17     return newSeqs;
18 Function mutate(seedSeq, i, type, tarNode):
19     seq ← seedSeq.clone();
20     seq.get(i).event.type ← type;
21     seq.get(i).event.tarNode ← tarNode;
22     seq.totalFaults ← seq.totalFaults + 1;
23     satisfy ← satisfyConstaints(seq, i);
24     if satisfy ∧ type.equals(CRASH) then
25         seq.deadNodes.add(tarNode);
26     else if satisfy ∧ type.equals(REBOOT) then
27         seq.deadNodes.remove(tarNode);
28     else
29         seq ← NULL;
30     return seq;
31 Function satisfyConstaints(s, i):
32     ftPt ← s.get(i);
33     e ← ftPt.event;
34     if e.type.equals(CRASH) ∧
        s.deadNodes.contain(e.tarNode) then
35         return false;
36     else if e.type.equals(REBOOT) ∧
        (s.deadNodes.contain(ftPt.node) ∨
        (!s.deadNodes.contain(e.tarNode))) then
37         return false;
38     else if e.type.equals(CRASH) ∧
        (s.deadNodes.size() ≥ MaxDeadNodes) then
39         return false;
40     else if s.totalFaults > MaxFaults then
41         return false;
42     return true;
```

generate a fault sequence $seedSeq$ according to the collected I/O points and injected faults after the fault injection testing. Any I/O points in $seedSeq$ that have new call stacks compared with I/O points in $testSeq$, will be identified as the I/O points executed during the recovery process. Therefore, when performing normal mutations on $seedSeq$, the newly injected faults on such I/O points will be marked as faults occurring during recovery. For example, as shown in Fig. 3, the fault point $FtPt^5_{n3}$ in $seedSeq$ has new call stack compared with all the I/O points in $testSeq$. Therefore, CrashFuzz takes $FtPt^5_{n3}$ as an I/O point executed during the recovery process. And the fault sequences $newSeq_3$ and $newSeq_4$ that inject faults on $FtPt^5_{n3}$, have a chance to be tested preferentially.

Third, for a group of fault sequences $newSeqs$ that are mutated from the same fault sequence $seedSeq$, CrashFuzz preferentially tests the sequences that inject faults on new I/O points. For example, in $newSeqs$, a fault sequence seq injects a fault on an I/O point p, which is new for tested fault sequences in $newSeqs$. Then, seq will be preferentially tested next compared with other untested fault sequences in $newSeqs$. Note that from the global perspective, the fault on the I/O point p may have been already tested. For example, a fault sequence seq' that is mutated from another fault sequence $seedSeq'$, may also inject a fault on p. And seq' has been tested. However, seq and seq' may face different system states when injecting a fault on p, because the previously injected faults in these two fault sequences are different. Thus, seq may still trigger new recovery behaviors.

The probability of taking above three types of fault sequences as suspicious candidates is configurable. If no fault sequence matches these three types, or these three types of fault sequences are not considered probabilistically, CrashFuzz directly takes all the fault sequences in the queue as the candidate fault sequences.

2) Picking a Fault Sequence to Test: For the selected candidate fault sequences, we compute a priority score for each of them. The priority scores are used to accelerate the testing process and test crash scenarios with multiple node crashes and reboots faster. Based on the priority scores, we randomly pick a fault sequence from the candidates to test.

We consider all the metrics shown in Table I to compute a priority score. Among them, the first three metrics (i.e., execution time, code coverage and waiting round) are adopted from existing fuzzing techniques, e.g., AFL [23]. The number of faults metric is specific to crash recovery bugs, and is inspired by a previous study on crash recovery bugs [8].

We prefer to test the fault sequences that are prone to have shorter execution time and larger code coverage first. Thus, we can test as many fault sequences as possible. For a fault sequence $newSeq$ mutated from $seedSeq$, the metrics of

TABLE I
METRICS IN PRIORITY SCORES

Metrics	Explanations
Execution time	The shorter the execution time, the higher the score.
Code coverage	The more code coverage, the higher the score.
Waiting round	The scores will be improved for the sequences that have long waiting time.
The number of faults	If the number of faults $\leq$ 6, the higher the number of faults, the higher the score; If the number of faults $>$ 6, the higher the number of faults, the lower the score.

$newSeq$ are obtained from the seed sequence $seedSeq$. After a fault injection test, we can obtain a fault sequence $seedSeq$, and get $seedSeq$'s execution time and code coverage according to the test results. For the new fault sequences mutated from $seedSeq$, we set them to have the same execution time and code coverage with $seedSeq$.

We prioritize the fault sequences that have waited too long according to the waiting round metric to avoid local optimum. The waiting round means the number of rounds a fault sequence waits in the queue before being tested.

We use the number of faults metric to test fault sequences with multiple faults faster, and avoid testing sequences that have too many faults. According to the empirical study of crash recovery bugs in cloud systems [8], the combination of no more than three node crashes and no more than three node reboots can trigger most of the crash recovery bugs. Therefore, for the fault sequences that contain no more than six faults, CrashFuzz preferentially tests the sequences that contain more faults to enable CrashFuzz to explore the combination of multiple faults faster. For the fault sequences that contain more than six faults, CrashFuzz gradually reduces their priorities. This is because the fault sequences that contain too many faults are not likely to trigger new crash recovery behaviors, but will increase the test time.

The values and metrics used in our fault sequence selection process can be tuned for different systems. We leave this for future work.

C. Testing Fault Sequences

For a fault sequence to be tested, CrashFuzz drives the workload to run the target system, collects runtime information, and controls system execution based on the collected information. In the paper, we only need to provide one fault-free workload for a target system to drive the test.

Collecting runtime information. During a fault injection test, CrashFuzz collects runtime information of every executed I/O point and reports the information to the fault injection controller for fault injection decisions. We also store the executed I/O points, injected faults, coverage information, execution time and other runtime feedbacks into files for the follow-up fault sequence generation and selection.

Controlling system execution. For a fault sequence $testSeq$ under test, CrashFuzz injects faults to the target system in order according to $testSeq$. During a fault injection test, the fault injection controller collects reports from every node in the cloud system, and compares the reported I/O points with the I/O points that require to inject faults in $testSeq$. If a reported I/O point matches the I/O point where the fault currently waiting to be injected in $testSeq$, the fault injection controller injects a corresponding node crash or reboot through predefined fault injection scripts. Otherwise, CrashFuzz informs the system to continue to execute. Note that different from distributed system model checkers [15]–[17], CrashFuzz does not control the execution of all the I/O points. It only focuses on the execution of I/O points that require to inject faults.

D. Detecting Crash Recovery Bugs

Similar to existing approaches, e.g., CoFI [27] and FCatch [18], we use some predefined checkers to find whether the target system goes into an unexpected state (i.e., a bug happens) and use a user-specified timeout threshold to detect hang bugs. For a target system and a workload, we implement specific checkers to detect following failure symptoms.

- General failures, e.g., ERROR entries in execution logs, unexpected node downtime, and system hangs. At the end of a test run, we check these general failures.
- Operation-specific failures, e.g., returning error code and reading stale data. These failures are checked when we run the workload.

For a fault sequence $testSeq$ under test, when the fault injection test completes or the test times out, we first check whether all the faults in $testSeq$ have been successfully injected within timeout period. If not, e.g., some I/O points did not appear due to nondeterminism, we add $testSeq$ back to the queue and go to the fault sequence selection stage (Section IV-B). Then, if the test does not finish within timeout period, we will test $testSeq$ right again with a larger timeout period. We report a hang bug if the test still cannot complete on time and go to the fault sequence selection stage (Section IV-B). If the test does not pass the checkers, we report a crash recovery bug and go to the fault sequence selection stage (Section IV-B). Finally, for the other cases, we go to the fault sequence generation and mutation stage (Section IV-A).

V. Implementation

Coverage collection. By identifying function entries, function exits and branches through ASM [24], CrashFuzz uniquely identifies every basic block (i.e., a straight-line sequence of code with only one entry point and one exit) of the target system and utilizes a 64KB byte array to store code coverage information. When a basic block is executed by the target system, CrashFuzz sets the corresponding position of the basic block in the byte array to 1.

I/O point identification. CrashFuzz identifies all the I/O points executed during a test run by instrumenting all the byte codes running in Java Virtual Machine (JVM). In our experiments, CrashFuzz intercepts I/O operations performed at the application level by tracking the usage of special APIs, e.g., native write APIs in `FileOutputStream` for file write operations, Remote Procedure Calls (RPCs) used for node communications in HBase and HDFS, serialize/deserialize APIs in class `Record` which are used for all socket messages in ZooKeeper. In addition, CrashFuzz also supports tracking I/O operations at JRE (Java Runtime Environment) level without modifying target systems, e.g., tracking native write APIs in `SocketOutputStream` for blocking socket messages and tracking write APIs in `SocketChannelImpl` for non-blocking socket messages. Intercepting I/O operations at JRE level can get more I/O operations and cause larger crash scenario space, but it makes CrashFuzz more general.

TABLE II
CLOUD SYSTEMS UNDER TEST

Cloud system	Workload
HDFS-3.3.1	Check safe mode, put/move/truncate/read/write file
HBase-2.4.8	Create/read/update/truncate/delete table
ZooKeeper-3.6.3	Create/read/update/delete znodes

VI. EVALUATION

Our evaluation aims to answer two research questions:

- **RQ1:** How effectively can CrashFuzz detect crash recovery bugs in real-world cloud systems? (Section VI-B)
- **RQ2:** How does CrashFuzz compare with other approaches for injecting node crashes/reboots in cloud systems? (Section VI-C)

A. Target Cloud Systems and Workloads

We evaluate CrashFuzz on three popular open-source cloud systems: the distributed key-value store ZooKeeper v3.6.3 [22], the distributed file system HDFS v3.3.1 [20] and the distributed NoSQL store HBase v2.4.8 [21]. These three target systems are complex. ZooKeeper, HBase and HDFS contain 34,496, 805,237 and 471,335 lines of Java code, respectively.

For each cloud system, we design a workload to drive the test. As shown in Table II the three workloads consist of cluster startup operations, common user operations and admin operations. Our workloads are complex. For example, running HDFS on a 5-node cluster without injecting any faults can produce around 400 I/O points for each node.

The target systems are deployed in several virtual machines with Docker 19.03.3. We build a distributed ZooKeeper cluster with 5 nodes, which can tolerate simultaneous downtime of up to 2 nodes; a distributed HBase cluster with 2 master nodes and 3 slave nodes, which can tolerate simultaneous downtime of at most 1 master node and downtime of at most 2 slave nodes; a distributed HDFS cluster with 2 master nodes and 3 slave nodes, which can tolerate simultaneous downtime of at most 1 master node and 1 slave node.

B. Effectiveness of Crash Recovery Bug Detection

1) Methodology: To evaluate CrashFuzz's effectiveness in revealing crash recovery bugs, we apply CrashFuzz on our target systems. We limit CrashFuzz to run at most 48 hours and set the maximum number of faults in a fault sequence as 10.

2) Overall Results: As shown in Table III, CrashFuzz has detected five crash recovery bugs in total, including 4 unknown bugs and 1 known bug. The bugs can cause cluster out of service, data loss, data staleness, operation failure and misleading error message. All the bugs have been reported to the developers. Among them, three bugs detected by CrashFuzz require injecting two faults.

3) Bug Study: The triggering process for bug HBASE-26883 is relatively complex. The HBase cluster used in our experiment consists of two master nodes (i.e., $master1$ and $master2$) and three slave nodes (i.e., $slave1$, $slave2$ and $slave3$). When the cluster starts up, $master1$ becomes the active master node, while $slave3$ becomes the meta slave node that holds the meta-data region. The client first sets the state of a table $mytable$ to `DISABLED`, and then requests the system to truncate $mytable$. The truncate process in HBase first deletes $mytable$, and then recreates a new empty table. After $master1$ successfully deleted the old $mytable$ and created new data regions for the new $mytable$, and before updating the information of the new table to the meta-data region, $master1$ crashes. Subsequently, the meta slave node $slave3$ also crashes. In the recovery process, the backup master node $master2$ is activated to recover the unfinished operations on $master1$. However, $master2$ accidentally deletes $mytable$.

For bug ZOOKEEPER-4503, a client creates an ephemeral znode "/eph". After a while, the follower node $zk1$ that maintains the session with the client crashes. During $zk1$'s downtime, the ZooKeeper cluster deletes the ephemeral znode "/eph" since its corresponding session has been disconnected. When $zk1$ is restarted, another client happens to initiate another session with $zk1$ and read the value of "/eph". This read operation occurs before $zk1$ completes synchronizing with the leader. Then the client unexpectedly gets the stale value of "/eph". A developer indicates that the ZooKeeper server should not start serving client requests until the synchronization is finished. But this bug shows that there is something wrong in this not easy-to-follow part in ZooKeeper. A node crash and a node reboot are both required for triggering this bug.

For bug HBASE-26897, the active master node and the meta slave node in HBase crash at the same time. In this crash scenario, the backup master node is blocked in the startup process to wait for an available meta region forever.

For bug HBASE-26370, a client requests to truncate a table. In this process, the active master node crashes after it marked the table to be truncated as `ENABLING`, and before it completes the truncate process. This makes the client receive a `TableNotDisabledException`, while the table has already been truncated by the backup master node in the recovery process. The client should not be disturbed by the unexpected exception.

For bug HDFS-16508, there are two master nodes, i.e., $master1$ and $master2$ in the HDFS cluster. When starting the cluster, $master1$ crashes at the very beginning, and $master2$ transfers to be active. This node crash can be tolerated and the system can provide services normally. However, when the client commands to get safe mode of the system, the dfsadmin command fails due to an IOException: cannot connect to $master1$. This is a duplicated bug that has already been reported in other HDFS versions.

C. Comparison with Alternative Fault Injection Approaches

1) Methodology: We compare CrashFuzz with three alternative fault injection approaches as shown in Table IV.

CrashFuzz$^-$. CrashFuzz$^-$ adopts first-in first-out (FIFO) strategy when picking a fault sequence from the fault sequence queue. Except for the fault sequence selection strategy, other components of CrashFuzz$^-$ are the same as CrashFuzz. By

TABLE III
CRASH RECOVERY BUG DETECTION RESULTS

	Bug ID	Failure Symptoms	# of Faults	Random	BruteForce	CrashFuzz$^-$	CrashFuzz
Unknown Bugs	HBASE-26883	Data loss	2	✗	✗	✗	✓
	ZOOKEEPER-4503	Data staleness	2	✗	✗	✗	✓
	HBASE-26897	Cluster out of service	2	✗	✗	✗	✓
	HBASE-26370	Misleading error message	1	✓	✓	✓	✓
Known Bugs	HDFS-16508	Operation failure	1	✗	✓	✓	✓

TABLE IV
SETTINGS FOR ALTERNATIVE APPROACHES

Approach	Crash/Reboot Injection Point	Fault Sequence Generation	Fault Sequence Selection
CrashFuzz	I/O Points	Coverage Guided	Prioritization
CrashFuzz$^-$	I/O Points	Coverage Guided	FIFO
BruteForce	I/O Points	Enumeration	FIFO
Random	Random Time	Random	Random

comparing CrashFuzz$^-$ with CrashFuzz, we can evaluate how CrashFuzz's prioritization strategy contributes to bug detection and code coverage.

BruteForce. Similar to FATE [12], BruteForce adopts an enumeration strategy that systematically explores all possible combinations of node crashes and reboots (FATE supports node crashes, but does not support node reboots). Specifically, BruteForce first tests all the fault sequences with one fault, and then test all the sequences with two faults, and so on.

Random. Random first decides the number of total faults to be injected randomly, i.e., N. Then, Random generates a fault sequence that contains N node crashes and reboots. Random generates faults in a fault sequence one by one. Specifically, when generating the ith fault, Random randomly chooses a fault type (i.e., node crash or node reboot) first. For the selected fault type, Random randomly selects a node in the target system as the target node. The ith fault happens at a random time $waitTime_i$ between $[0, aveTime - \sum_{j=1}^{i-1} waitTime_j)$ after the injection of $(i-1)$th fault. Here, $aveTime$ is the average testing execution time.

BruteForce and Random only keeps the fault sequences that satisfy the same constraints as that in CrashFuzz for a fair comparison. All the above three approaches are performed with the same settings as CrashFuzz, e.g., cluster configurations, the maximum number of faults and the total testing time. In addition, all the approaches use the same bug checkers as that in CrashFuzz for bug detection.

We do not compare CrashFuzz with other fault injection approaches [12], [18], [19] for the following reasons. (1) Their tools are unavailable. (2) FCatch [18] and CrashTuner [19] focus on specific fault scenarios and cannot systematically test the cloud system. (3) FATE [12] runs in a brute-force mode by default, which is similar to BruteForce used in our experiment. The strategies supported by FATE to reduce fault scenarios require special information and are not easy to reimplement.

2) Comparison Results: To evaluate CrashFuzz's effectiveness, we compare CrashFuzz with the alternative approaches from four aspects: (1) Effectiveness in detecting crash recovery bugs. (2) Effectiveness in testing crash recovery behaviors. (3) Proportion of valuable tests that contribute to code coverage. (4) The number of tested fault sequences.

Bug detection. As shown in Table III, compared with alternative approaches, three out of five crash recovery bugs can only be detected by CrashFuzz. Bug HBASE-26370 can be detected by all the four fault injection approaches since it has a relatively large bug triggering time window. Bug HDFS-16508 was not detected by Random. Because the bug is triggered by injecting a node crash at the very beginning of the cluster startup. Compared with Random, other approaches have a relatively higher possibility to expose it. All the three alternative approaches did not reveal any new bugs for CrashFuzz. Therefore, CrashFuzz is more effective in revealing crash recovery bugs compared with alternative approaches.

Code coverage. We measure CrashFuzz's effectiveness in testing recovery behaviors of cloud systems through the overall code coverage in 48 hours. Fig. 4 shows how the overall code coverage varies over time for CrashFuzz and alternative approaches. The x-axis represents the test time (hours), and the y-axis represents the overall code coverage over time.

Note that we cannot compute the crash recovery code coverage, because we cannot distinguish crash recovery code from regular code in complex cloud systems. Our studied cloud systems are complex and contain large amounts of code, e.g., HBase contains 805,237 lines of Java code. Cloud systems have to handle various crash recovery scenarios, and their regular logic and crash recovery logic are usually mixed up together. For example, in HBase, region reassign logic is invoked by both user requests and crash recovery for a slave node. Therefore, we (not system developers) cannot distinguish crash recovery code from regular code.

Since we cannot distinguish crash recovery code from regular code in complex cloud systems, covering more code is not necessarily covering more crash recovery code. For a cloud system, we use the same workload for all the fault injection testing, injecting crashes and reboots can cause new behaviors, which are very likely to be crash recovery behaviors. Therefore, the increased code coverage in our experiment is more likely to be caused by the crash recovery code.

For CrashFuzz, after running 48 hours, ZooKeeper, HBase and HDFS reach an overall coverage of 14.30%, 22.95% and 23.03%, respectively. This appears to be low. However, it is reasonable, since we only perform fault injection testing for a single workload in each target system. Large portions of these cloud systems are not tested yet.

As shown in Fig. 4, CrashFuzz$^-$ reaches 14.30%, 22.92%

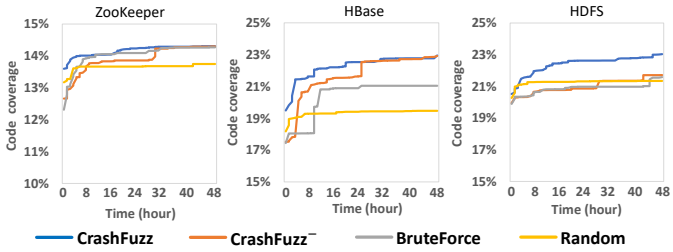

Fig. 4. Overall code coverage.

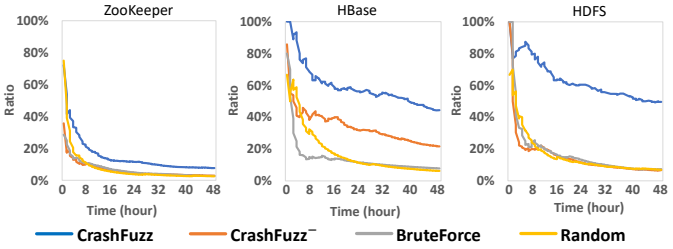

Fig. 5. The proportion of the tests that increase code coverage.

and 21.71% code coverage for ZooKeeper, HBase and HDFS respectively. BruteForce reaches 14.27%, 21.04% and 21.58% code coverage for ZooKeeper, HBase and HDFS, respectively. Random reaches 13.74%, 19.47% and 21.34% code coverage for ZooKeeper, HBase and HDFS, respectively. Compared with alternative approaches, CrashFuzz covers the most code at the end of the test in all the three target systems. Meanwhile, CrashFuzz can achieve higher code coverage faster.

We can see that CrashFuzz and CrashFuzz$^-$ cover almost the same amount of code for HBase and ZooKeeper. While for HDFS, CrashFuzz covers more code than alternative approaches. This is because HDFS has a larger crash scenario space than HBase and ZooKeeper. Without injecting any faults, running a workload for HDFS can produce around 2,000 I/O operations, while HBase and ZooKeeper execute around 600 I/O operations, respectively. Specifically, after testing about 8 hours, the code coverage of Random tends to be stable in the three cloud systems. Compared with other approaches, Random is less effective in testing cloud systems.

Valuable tests. Fig. 5 shows the proportion of the tests that were successfully triggered and contribute to the overall code coverage. The x-axis represents the test time (hours), and the y-axis represents the ratio. As the test time increases, for all the approaches, the proportion of the successfully triggered tests that increase the code coverage decreases gradually. Compared with alternative approaches, CrashFuzz has the slowest decline speed. This indicates that CrashFuzz is more likely to test fault sequences that can increase the overall code coverage.

Tested fault sequences. We record the number of fault sequences successfully triggered during the test in Table V. The values in parentheses show the numbers of fault sequences that contribute to code coverage. In total, CrashFuzz successfully tests 603 fault sequences in ZooKeeper, 142 fault sequences in HBase and 143 fault sequences in HDFS, respectively.

TABLE V
SUCCESSFULLY TRIGGERED FAULT SEQUENCES

	CrashFuzz	CrashFuzz$^-$	BruteForce	Random
ZooKeeper	603 (46)	1309 (38)	1324 (35)	595 (15)
HBase	142 (63)	494 (107)	371 (29)	208 (13)
HDFS	143 (71)	599 (40)	599 (44)	303 (21)

The values in parentheses show the numbers of fault sequences that can increase the code coverage.

TABLE VI
RUNTIME OVERHEAD

System	Baseline	Info Collection	Average Test Time
ZK	7s	3.6X	14.1X
HB	44s	3.1X	15.4X
HDFS	71s	2.5X	8.0X

For the alternative approaches, CrashFuzz$^-$ and BruteForce test more fault sequences than CrashFuzz and Random. Because these two approaches adopt FIFO method to pick fault sequences for testing. The fault sequences tested by these two approaches contain at most two faults in 48 hours. While CrashFuzz injects at most six faults for ZooKeeper, and injects at most five faults for HBase and HDFS. Random injects at most 10 faults for the three cloud systems. The fault sequences that contain more faults will trigger additional recovery behaviors and thus increase the testing time. Our experimental results show that the fault sequences that contain multiple faults can still increase the code coverage. CrashFuzz can explore the crash scenarios containing multiple faults faster, which implies crash recovery bugs that only manifest under multiple faults can be revealed faster.

D. Runtime Overhead

Table VI shows the runtime overhead of CrashFuzz. The **Baseline** column shows the original workload run time without any instrumentation, the **Info Collection** column shows the average fault-free run time of the workload running on the instrumented system compared to the baseline run time, and the **Average Test Time** column shows the average run time of a fault injection test compared to the baseline run time.

The results show that CrashFuzz introduces 2.5X to 3.6X overhead for run time system information collection. On average, testing a fault sequence takes around 8.0X to 15.4X baseline time. This is because each test includes a series of operations such as initializing the execution environment (e.g., preparing the initial cluster state), running the workload, handling the injected faults, collecting runtime information, checking failure symptoms, and so on.

VII. DISCUSSION

We now discuss CrashFuzz's limitations and potential threats.

A. Limitations

Workloads. Crash recovery bugs usually require proper workloads to be triggered. However, it is challenging to

automatically construct good workloads for cloud systems. Different cloud systems provide different complex APIs. There is no general way to construct workloads for different cloud systems. Fortunately, developers have developed a lot of test cases, which can be used as workloads in our test. CrashFuzz can support more workloads easily.

Generating fault sequences. CrashFuzz only concerns about fault sequences that are expected to be tolerated by cloud systems. Hence, CrashFuzz may miss some crash recovery bugs whose manifestation requires to fail the whole cloud systems. Even though CrashFuzz does not consider such crash scenarios, our evaluation shows that CrashFuzz is effective in testing cloud systems. We will extend CrashFuzz to support such crash scenarios in our future work.

Testing fault sequences. Similar to FATE [12], when performing fault injection testing, CrashFuzz does not control the execution order of all the I/O points. It only focuses on the execution order of I/O points that need to inject faults. Thus, we cannot guarantee cloud systems can run exactly the same as we expect due to the nondeterminism. This poses challenges for bug diagnosis. Besides, CrashFuzz does not reorder concurrent I/O operations like distributed system model checkers [15]–[17], [25]–[27]. This may miss concurrent bugs that lie in crash recovery code.

B. Threats to Validity

We evaluate CrashFuzz on the latest versions of three popular cloud systems. Therefore, our experimental results may not reflect the situation in other cloud systems. However, we strive to be unbiased by selecting systems with different functionalities (i.e., a distributed file system, a distributed NoSQL database and a distributed synchronization service) and various crash recovery mechanisms (e.g., automatic failover, synchronization, replication and so on).

We deploy the target cloud systems on a five-node cluster in our evaluation, respectively. While real-world cloud systems may contain more nodes. However, a study on crash recovery bugs [8] shows, 97% of crash recovery bugs involve four nodes or fewer. Therefore, it is reasonable to conduct our evaluation for the target systems on a cluster with five nodes. If there are more nodes in a cloud system, more I/O operations will be produced and thus increase the crash scenario space. However, the increased I/O operations are more likely to be the operations that have already appeared in a small scale cluster. For example, in a ZooKeeper cluster, all follower nodes share the same protocol. Thus, adding more follower nodes are unlikely to increase new crash scenarios. In our approach, we prioritize *new* crash scenarios to test cloud systems instead of enumerating all crash scenarios. Therefore, we can still perform an effective test for a cloud system with more nodes.

VIII. Related Work

Fault injection frameworks. The random fault injection frameworks [10], [28]–[30] can randomly inject node crashes and other types of faults in cloud systems. These frameworks are relatively simple to implement, but they are ineffective in testing cloud systems [31]. Frameworks like PreFail [13] that enable developers to write their own fault injection strategies. FATE [12] uses the brute-force search to enumerate all combinations of multiple faults. Other fault injection approaches [14], [19], [32]–[34] either focus on special crash scenarios or cannot be used for injecting node crashes/reboots.

Distributed system model checkers. Distributed system model checkers can also be used to expose crash recovery bugs. These works intercept non-deterministic events including node crashes and permute their orders [15]–[17], [25]–[27]. However, these model checkers do not only focus on crash recovery bugs. Therefore, they have to explore a lot of crash-unrelated states until exposing a crash recovery bug. All of them suffer from state space explosion problems for real-world cloud systems, especially when considering multiple node crashes and reboots.

Cloud bug detection. FCatch [18] predicts time-of-fault bugs by observing possible conflicting operations under crashes. Zhang et al. propose a testing framework and static checkers to reveal upgrade failures [35]. Deminer [36] detects crash recovery bugs that are triggered by node crashes occurring between related I/O operations. DisTA [37] supports dynamic taint tracking for cloud systems. Some works have been conducted on detecting crash-unrelated bugs, e.g., cloud concurrency bugs [38], [39], performance cascading bugs [40], data-corruption related hang bugs [41], wrong exception handling [42]–[44], and so on. These works either aim to detect bugs that are not related to node crashes/reboots, or can only cover limited crash scenarios.

Fuzzing. Fuzzing is an automated testing technique that can detect vulnerabilities by randomly generating a lot of inputs [45]. Many fuzzing approaches are proposed to stress real-world programs [23], [46]–[54]. GFuzz [55] detects channel-related concurrency bugs in Go programs by mutating the processing orders of concurrent messages. FIFUZZ [56] detects bugs hidden in error handling code by mutating different combinations of errors. These approaches do not focus on fuzzing node crashes and reboots for cloud systems.

IX. Conclusion

We propose CrashFuzz, a novel coverage guided fault injection approach to inject node crashes and reboots for a cloud system, and systematically test if the cloud system can correctly recover from various crash scenarios. The core idea of CrashFuzz is generating and mutating crash scenarios according to runtime feedbacks, e.g., coverage and I/O information. The evaluation on three popular cloud systems shows that CrashFuzz can detect more crash recovery bugs and achieve higher code coverage than alternative approaches.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62072444, U20A6003, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences.

REFERENCES

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 137–149.

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 205–218.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 205–220.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 29–43.

[5] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 295–308.

[6] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache Hadoop YARN: Yet another resource negotiator," in *Proceedings of Annual Symposium on Cloud Computing (SoCC)*, 2013, pp. 1–16.

[7] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 335–350.

[8] Y. Gao, W. Dou, F. Qin, C. Gao, D. Wang, J. Wei, R. Huang, L. Zhou, and Y. Wu, "An empirical study on crash recovery bugs in large-scale distributed systems," in *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 539–550.

[9] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou, "Failure recovery: When the cure is worse than the disease," in *Proceedings of USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2013, pp. 1–6.

[10] (2016) Jepsen. [Online]. Available: https://github.com/jepsen-io/jepsen

[11] (2012) Chaos Monkey. [Online]. Available: https://netflix.github.io/chaosmonkey

[12] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011, pp. 238–252.

[13] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A programmable tool for multiple-failure injection," in *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011, pp. 171–188.

[14] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018, pp. 51–68.

[15] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "MODIST: Transparent model checking of unmodified distributed systems," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009, pp. 213–228.

[16] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi, "SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 399–414.

[17] J. F. Lukman, H. Ke, C. A. Stuardo, R. O. Suminto, D. H. Kurniawan, D. Simon, S. Priambada, C. Tian, F. Ye, T. Leesatapornwongsa *et al.*, "FlyMC: Highly scalable testing of complex interleavings in distributed systems," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2019, pp. 1–16.

[18] H. Liu, X. Wang, G. Li, S. Lu, F. Ye, and C. Tian, "FCatch: Automatically detecting time-of-fault bugs in cloud systems," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018, pp. 419–431.

[19] J. Lu, C. Liu, L. Li, X. Feng, F. Tan, J. Ynag, and L. You, "CrashTuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2019, pp. 114–130.

[20] (2008) Apache Hadoop HDFS. [Online]. Available: http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[21] (2007) Apache HBase. [Online]. Available: https://hbase.apache.org/

[22] (2010) Apache ZooKeeper. [Online]. Available: https://zookeeper.apache.org/

[23] M. Zalewski. (2020) American Fuzzy Lop. [Online]. Available: https://lcamtuf.coredump.cx/afl/

[24] (2005) ASM. [Online]. Available: https://asm.ow2.io/

[25] H. Guo, M. Wu, L. Zhou, G. Hu, J. Yang, and L. Zhang, "Practica software model checking via dynamic interface reduction," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 265–278.

[26] J. Simsa, R. Bryant, and G. Gibson, "dBug: Systematic evaluation of distributed systems," in *Proceedings of International Conference on Systems Software Verification (SSV)*, 2010, pp. 1–9.

[27] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat, "Life, death, and the critical transition: Finding liveness bugs in systems code," in *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2007, pp. 243–256.

[28] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proceedings of Annual ACM Symposium on Principles of Distributed Computing (PoDC)*, 2007, pp. 398–407.

[29] A. Henry, "Cloud storage FUD: Failure and uncertainty and durability," in *Proceedings of USENIX Symposium on File and Storage Technologies (FAST)*, 2009.

[30] T. Hoff. (2010) Netflix: Continually test by failing servers with Chaos monkey. [Online]. Available: http://highscalability.com

[31] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proceedings of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2015, pp. 331–346.

[32] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2017, pp. 149–165.

[33] H. Chen, W. Dou, D. Wang, and F. Qin, "CoFI: Consistency-guided fault injection for cloud systems," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 536–547.

[34] R. Alagappan, A. Ganesan, Y. Patel, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Correlated crash vulnerabilities," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 151–167.

[35] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan, "Understanding and detecting software upgrade failures in distributed systems," in *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2021, pp. 116–131.

[36] Y. Gao, D. Wang, Q. Dai, W. Dou, and J. Wei, "Common data guided crash injection for cloud systems," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE Demo)*, 2022, pp. 36–40.

[37] D. Wang, Y. Gao, W. Dou, and J. Wei, "DisTA: Generic dynamic taint tracking for java-based distributed systems," in *Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022, pp. 547–558.

[38] H. Liu, G. Li, J. F. Lukman, J. Li, S. Lu, H. S. Gunawi, and C. Tian, "DCatch: Automatically detecting distributed concurrency bugs in cloud systems," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017, pp. 677–691.

[39] X. Yuan and J. Yang, "Effective concurrency testing for distributed systems," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020, pp. 1141–1156.

[40] J. Li, Y. Chen, H. Liu, S. Lu, Y. Zhang, H. S. Gunawi, X. Gu, X. Lu, and D. Li, "PCatch: Automatically detecting performance cascading bugs in cloud systems," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2018, pp. 1–14.

[41] T. Dai, J. He, X. Gu, S. Lu, and P. Wang, "DScope: Detecting real-world data corruption hang bugs in cloud server systems," in *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, 2018, pp. 313–325.

[42] S. Saha, J.-P. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *Proceedings of Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2013, pp. 1–12.

[43] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm, "Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 249–265.

[44] H. Chen, W. Dou, Y. Jiang, and F. Qin, "Understanding exception-related bugs in large-scale cloud systems," in *Proceedings of IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 339–351.

[45] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[46] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *Proceedings of USENIX Conference on Security Symposium (SECURITY)*, 2018, pp. 745–761.

[47] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2013, pp. 197–208.

[48] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 206–215.

[49] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594.

[50] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

[51] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *Proceedings of IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[52] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 475–485.

[53] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2019, pp. 724–735.

[54] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2019, pp. 329–340.

[55] Z. Liu, S. Xia, Y. Liang, L. Song, and H. Hu, "Who goes first? Detecting Go concurrency bugs via message reordering," in *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2022, pp. 888–902.

[56] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Fuzzing error handling code using context-sensitive software fault injection," in *Proceedings of USENIX Conference on Security Symposium (SECURITY)*, 2020, pp. 2595–2612.