Common Data Guided Crash Injection for Cloud Systems

Yu Gao
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
gaoyu15@otcaix.iscas.ac.cn

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

Qianwang Dai
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
daiqianwang19@otcaix.iscas.ac.cn

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

Jun Wei
State Key Lab of Computer Sciences, Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences, Beijing, China
wj@otcaix.iscas.ac.cn

ABSTRACT

Modern distributed systems are designed to tolerate node crashes. However, incorrect crash recovery mechanisms and implementations can still introduce crash recovery bugs, and hurt reliability and availability of cloud systems. In this paper, we present *Deminer*, a novel crash injection technique that automatically injects node crashes/reboots to effectively expose crash recovery bugs in cloud systems. We observe that, node crashes that interrupt the execution of related operations, which store common data to different places (i.e., different storage paths or nodes), are more likely to trigger crash recovery bugs. Based on this observation, Deminer first tracks the critical data usage in a correct run. Then Deminer identifies related operations and predicts error-prone crash points. Finally, Deminer tests the predicted crash points and checks whether the target system can behave correctly. We have evaluated Deminer on three widely-used cloud systems: ZooKeeper, HBase and HDFS. Deminer has detected 6 crash recovery bugs. A video demonstration of Deminer is available at https://youtu.be/jS6KBcYnTSM.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; **Software reliability**; **Software testing and debugging**.

KEYWORDS

Fault injection, crash recovery, cloud system, bug detection

*Wensheng Dou is also affiliated with Nanjing Institute of Software Technology and University of Chinese Academy of Sciences, Nanjing, China. Wensheng Dou is the corresponding author.

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516852

ACM Reference Format:
Yu Gao, Dong Wang, Qianwang Dai, Wensheng Dou, and Jun Wei. 2022. Common Data Guided Crash Injection for Cloud Systems. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516852

1 INTRODUCTION

Cloud systems are widely adopted by modern enterprises, e.g., Google and Alibaba, as scalable computing frameworks [12], storage systems [4, 13, 16], cluster management services [2] and synchronization services [9], to support reliable services for users. Unfortunately, components in cloud systems, i.e., nodes, can inevitably fail [11], which can hurt the reliability and availability of cloud systems. Although cloud systems are designed to tolerate node crashes, it is challenging for developers to design and implement bug-free crash recovery protocols that guarantee all the node crashes can be tolerated and recovered from. Incorrect crash recovery protocols and implementations can introduce crash recovery bugs [15], and cause severe consequences, e.g., node downtime, data staleness and operation failures.

Among all kinds of crash scenarios, crashes that can cause inconsistent system states, are more likely to trigger crash recovery bugs. We observe that a node in the cloud system can store a piece of data (we refer it as the common data) to multiple places outside the node, e.g., other nodes in the cluster or a storage system. Crashing the node will remove its in-memory states, while the data stored outside the node can still be accessed and affect system behaviors. A node crash that interrupts the execution of those storage operations that use common data can cause inconsistent system states, and further fail the subsequent recovery process.

Figure 1 illustrates a buggy crash scenario that interrupts the execution of storage operations that use common data from ZooKeeper. Here, a node follows and synchronizes with the current leader in startup process. The follower node first gets the newest transaction ID from the leader through message $\texttt{msg}_1$ (Line 3). Then it deserializes the snapshot from another message $\texttt{msg}_2$ for synchronization (Line 11). Both $\texttt{msg}_1$ and $\texttt{msg}_2$ transfer the common data,

```
1. InputArchive leaderIs;

2. void followLeader() {
     //receive the LEADERINFO which contains
     //the new epoch value from leader
3.   long newEpochZxid = registerWithLeader();    msg1
4.   syncWithLeader(newEpochZxid);
5. }

6. void syncWithLeader(long newEpochZxid) {
7.   QuorumPacket qp = new QuorumPacket();
8.   readPacket(qp);
9.   long newEpoch = getEpochFromZxid(newLeaderZxid);
10.  if (qp.getType() == Leader.SNAP) {
       //deserialize the snapshot that contains the
       //new epoch value from leader
11.    zkDatabase.deserializeSnapshot(leaderIs);  msg2
12.  }
13.  while (self.isRunning()) {
14.    readPacket(qp);
15.    switch (qp.getType()) {
16.    case Leader.NEWLEADER:
         //store current data state to snapshot file
17.      takeSnapshot();
         //CRASH
         //store new epoch to currentEpoch file
18.      setCurrentEpoch(newEpoch);
19.      break;
20.    }
21.  }
22.}
```

Figure 1: Simplified code snippet from ZooKeeper.

i.e., the new epoch value. When receiving the UPTODATE message from the leader (Line 16), the follower first takes a snapshot of the current data state (Line 17), and then stores the new epoch value to the currentEpoch file (Line 18). When the follower crashes between Line 17 and Line 18, it will never be able to start without human intervention due to the inconsistent epoch value stored in the snapshot file and the currentEpoch file.

Recent research has been conducted on detecting bugs related to node crashes. Random fault injection frameworks [3, 5] inject crashes randomly, and hope to hit small bug-triggering time windows through many tries. Distributed model checkers [7, 19, 24, 28] that systematically permute the orderings of non-deterministic distributed events, suffer from the state explosion problem. Bug detection approaches [22, 23] based on program analysis are designed to detect time-of-fault bugs and meta-info related bugs, respectively.

In this paper, we propose *Deminer*, a novel approach to expose crash recovery bugs. Deminer works by injecting node crashes and reboots between related operations that store the common data to different places, i.e., different nodes and storage paths. First, Deminer tracks how the critical data are used by observing a correct run. Second, Deminer identifies related operations that use common data based on the execution trace, and predicts error-prone crashes that can interrupt the execution of these related operations. Finally, Deminer injects the predicted crashes/reboots to the target system and checks failure symptoms to confirm harmful crashes.

We have implemented a tool of Deminer, and applied it to the latest versions of three popular cloud systems: ZooKeeper, HBase and HDFS. Deminer has detected 6 crash recovery bugs. These detected bugs can cause operation failure, data staleness, node downtime, cluster out of service and and misleading error message.

2 DEMINER

Deminer first traces data usage of the target system at run time (Section 2.1), and then predicts error-prone crashes based on the execution trace (Section 2.2). Finally, Deminer injects predicted node crashes and corresponding node reboots to the target system and validates system behaviors (Section 2.3).

2.1 Data Usage Tracing

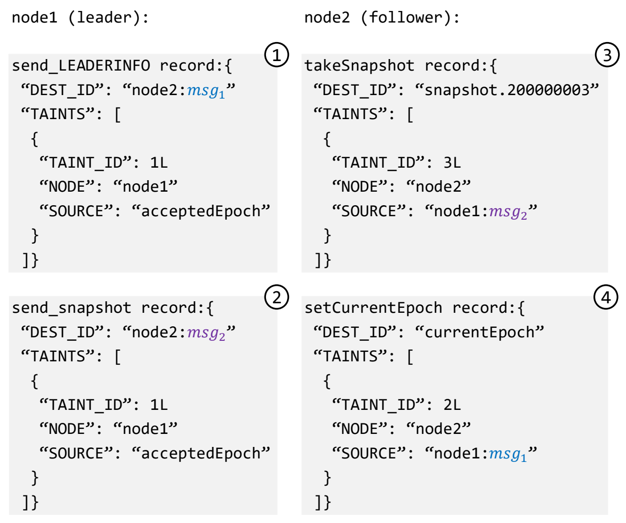
Deminer first tracks how the critical data flows at run time and trace storage operations that use the critical data. The critical data refers to the data read from a user specified storage path and the data received from a message sent by another node in the cluster or the client. Inconsistent cognition of these data is more likely to cause serious consequences. Specifically, a storage operation refers to an RPC call/socket sending operation that sends data to another node, or a file write operation that updates data to a storage path (e.g., a local file, an HDFS file or a ZooKeeper path).

Intra-node data tracking. Deminer tracks how the critical data flows in a node based on Phosphor [8], a dynamic taint tracking system for the Java Virtual Machine (JVM). Phosphor adds a shadow variable for each variable, and a shadow field for each object to store the corresponding taints of the data, and instruments all byte code that runs in the JVM to track taint propagation. As Phosphor is implemented within the JVM, we cannot track the data that pass through between nodes and local files.

Inter-node data tracking. Deminer performs inter-node data tracking by adding a random integer (we refer it as message ID) as an extra parameter/field for an RPC/socket message. For example, the LEADERINFO message received by the follower shown in Figure 1 (Line 3), is attached with a message ID $\texttt{msg}_1$. The snapshot message (Line 11) is attached with a message ID $\texttt{msg}_2$. By using the message ID, Deminer can pair a message sending point with its corresponding message receiving point. Therefore, Deminer can track inter-node data flows at off-line analysis stage (Section 2.2).

Once a piece of critical data is obtained by the system, Deminer generates a unique taint to mark the data. As shown in Figure 2, a taint contains three parts: (1) taint ID, we simply use a volatile long type number; (2) node ID, which uniquely identifies every node in the cloud system. We use a combination of server IP and the process identifier (PID); (3) data source, which specifies where the data comes from. We use the path string for the data read from a storage system, and use the node IP and the message ID for the data received from a message.

When a storage operation uses the critical data, Deminer generates a record. As shown in Figure 2, a record mainly consists of four pars: (1) the destination ID. We use the path string for the operation written to a storage system, and use the destination node IP and the message ID for an RPC call/socket sending operation; (2) taints, specifying the critical data used by the operation; (3) call stack; (4) local time stamp counter obtained by RDTSCP, which provides an



```
node1 (leader):                        node2 (follower):

send_LEADERINFO record:{           ①   takeSnapshot record:{              ③
 "DEST_ID": "node2:$msg_1$"              "DEST_ID": "snapshot.200000003"
 "TAINTS": [                             "TAINTS": [
  {                                       {
   "TAINT_ID": 1L                          "TAINT_ID": 3L
   "NODE": "node1"                         "NODE": "node2"
   "SOURCE": "acceptedEpoch"               "SOURCE": "node1:$msg_2$"
  }                                       }
 ]}                                      ]}

send_snapshot record:{             ②   setCurrentEpoch record:{           ④
 "DEST_ID": "node2:$msg_2$"              "DEST_ID": "currentEpoch"
 "TAINTS": [                             "TAINTS": [
  {                                       {
   "TAINT_ID": 1L                          "TAINT_ID": 2L
   "NODE": "node1"                         "NODE": "node2"
   "SOURCE": "acceptedEpoch"               "SOURCE": "node1:$msg_1$"
  }                                       }
 ]}                                      ]}
```

Figure 2: Simplified execution trace for Figure 1.

approximate nano-second level ordering among operations in one machine.

2.2 Crash Prediction

Deminer identifies related storage operation pairs that use common data based on the execution trace collected in Section 2.1, and generates crashes that interrupt the execution of related pairs.

Identifying related operation pairs. Deminer decides whether two storage operations are related through two steps. For two operations that are performed by the same node and have different destination ID, Deminer first checks whether their corresponding records have nonempty common taints. For example, in Figure 2, the record ① and ② have a common taint whose taint ID is `1L`, which denotes that the two operations use the common data. Thus, ① and ② are a pair of related operations.

If two records have empty common taints, e.g., the record ③ and ④, Deminer further checks if they use the common data obtained in another remote node. For record ③, it contains a taint whose source is `node1`:$\mathtt{msg}_2$, which means the taint is propagated from `node1` through message $\mathtt{msg}_2$. Based on the message ID $\mathtt{msg}_2$, Deminer can easily identify an internode data flow from record ② to record ③. Similarly, there also exists a data flow from record ① to record ④. Record ① and ② are related. Therefore, record ③ and ④ are also related since they both use the common data propagated from `node1`.

Predicting crashes. For a pair of related operations (rec_i, rec_j), in which rec_j's time stamp is larger than rec_i, Deminer generates a crash $[rec_i, C, rec_j]$, which denotes a crash C after the execution of the previous operation rec_i and right before the execution of the latter operation rec_j. For a group of crashes that have same latter operation rec_j, Deminer only keeps the crash whose previous operation rec_i has the largest time stamp.

2.3 Crash Injection Testing

In order to confirm real harmful crashes, Deminer automatically runs the workload used in the tracing phase under a triggering mode, tests the predicted crashes and checks failure symptoms to find real issues.

In a test run, only one crash is tested. Take the crash $[rec_i, C, rec_j]$ as an example. When the target system runs under triggering mode, Deminer collects the system runtime information at every storage operation point and checks whether the current node runs into the crash point. When the previous operation rec_i has been performed and the latter operation rec_j is about to execute, the node reports to the fault injection engine and waits for the engine's decision. For the first report received by the fault injection engine, it will directly kill the reporting node. After waiting for random time within one minute, Deminer reboots the node. For latter reports received by the fault injection engine, it will ignore them and inform the reporting node to continue execution. To combat the non-determinacy of the system execution, for every tested crash, Deminer tries to test it at most five times until the corresponding crash/reboot can be triggered.

For every tested workload, we implement specific checkers to detect failure symptoms. 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). Once a failure symptom manifests, Deminer will generate the corresponding bug report for further inspection. Uses can easily implement checkers for other workloads based on our checkers.

3 IMPLEMENTATION

Deminer uses ASM [1], a byte code manipulation library, to instrument the target system. Deminer supports three storage systems: local file system, HDFS distributed file system, and the key-value store ZooKeeper. For local file read/write operations, Deminer tracks the use of native JDK methods in class `FileInputStream`, `FileOutputStream` and `RandomAccessFile` that are used for accessing local files. For HDFS and ZooKeeper, Deminer tracks the use of their corresponding read/write client APIs.

For RPC, the latest versions of Hadoop and HBase implement RPC based on Protocol buffer. Deminer first obtains predefined Hadoop RPC library interfaces, and identifies HBase RPC library interfaces by checking whether the name of an interface is ended with `"Service$BlockingInterface"`. Then Deminer can easily identify RPC functions based on the RPC library interfaces. For socket, ZooKeeper uses a super-class `Record` for all socket messages. Deminer identifies socket sending and receiving based on how Record objects are used.

4 DEMINER USAGE

Deminer is implemented as a command line tool and supports JVM 1.8. Four main steps are needed when using Deminer:

1) **Generating an instrumented version of the runtime environment.** We can run the command `"java -jar Deminer.jar -forJava <jre_path> <output_path>"` to prepare an instrumented JRE.
   - `-forJava`: Specify the instrumentation for JRE.

- `jre_path`: Path for the input JRE.
- `output_path`: Path for the instrumented JRE.

2) **Run a workload at tracing mode.** We can configure every node of the target system to use the instrumented JRE and include the Deminer as the Java agent with a JVM argument. Take ZooKeeper as an example. We can modify `zkEnv.sh` file and add following configuration:

```
JAVA = <instrumented_jre_path>/bin/java
Deminer_JVMFLAGS =
-Xbootclasspath/a:<Deminer_path>/Deminer.jar
-javaagent:<Deminer_path>/Deminer.jar
=useTool=true,recordPhase=true,forZk=true,
jdkFile=true,recordPath=<trace_path>
```

The parameters are explained as follows:
- `useTool`: `true` for using Deminer.
- `recordPhase`: `true` for running system under tracing mode, and `false` for running system under triggering mode.
- `forZk`: `true` for tracking ZooKeeper socket messages.
- `jdkFile`: `true` for tracking reads/writes to local files.

Then we can use docker to deploy a target system cluster with the above configuration, and run a workload.

3) **Perform off-line analysis.** We can collect execution traces from every node to the directory `<trace_path>`, and use `node_ip` to specify the IPs of the nodes of the cluster (split with :). Then we can run the following command:

```
java -cp Deminer.jar <analysis_main_class>
    <trace_path> <node_ip> <output_path>
```

Deminer will generate identified related pairs and predicted crashes under `<output_path>`.

4) **Test crash points.** Now we can configure the target system to run under the triggering mode, and specify corresponding scripts, e.g., crash/reboot script and checker script, at a file `<cfg_file_path>`. We can run the following command on the host machine to start the testing process:

```
java -cp Deminer.jar <triggering_main_class>
    <listening_port> <cfg_file_path>
```

Finally, Deminer will generate a crash report for the crashes that do not pass checkers. Checkers used in our experiments contain at most 70 lines of code. Developers can easily implement checkers for other workloads based on our checkers.

5 EVALUATION

5.1 Methodology

To evaluate Deminer's effectiveness in detecting crash recovery bugs, We implement seven workloads and evaluate Deminer on the latest versions of three widely-used open-source distributed systems (shown in Table 1): ZooKeeper distributed synchronization service, HDFS distributed file system and HBase distributed key-value stores. These three systems can represent different kinds of cloud systems and implement different crash recovery mechanisms. The workloads involve the startup process of the target system, common admin operations (e.g., NameNode failover in HDFS) and user operations (e.g., file movement in HDFS). The workloads are running on the clusters with minimal nodes that are expected to tolerate a node crash and a node reboot without fault.

Table 1: Experimental settings for target systems

System	Workload
HDFS-3.3.1	1. Put/move file, read/write file 2. Read/write file + failover NameNode
HBase-2.4.8/ HBase-1.7.1	1. Create/read/update/truncate/delete table 2. Create/read/delete table + failover HMaster 3. Create/read table + failover meta HRegionServer
ZooKeeper-3.6.3	1. Create/read/update/delete znodes 2. Create/update znode + failover leader node

Table 2: Bugs triggered by Deminer.

Bug ID	Failure Symptom	Reboot
hb26370	Misleading error message	✗
hb26391	Data staleness	✗
hb26420	Cluster out of service	✗
zk4283	Node downtime	✓
zk4416	Node downtime	✓
hd16381	Operation failure	✗

We deploy the target systems in several virtual machines with Docker 19.03.3. The guest OS in VM uses Ubuntu 20.04 and JVM 1.8. The host machine is equipped with a 64-bit CentOS Linux release 7.3.1611, JVM 1.8, two 16-core 2.10GHz Intel(R) Xeon(R) Gold 6130 CPUs, and 125 GB of RAM.

5.2 Bug Detection Results

As shown in Table 2, Deminer has identified six crash recovery bugs, including three bugs from HBase, one bug from HDFS and two bugs from ZooKeeper. We can also see that two bugs require node reboot to expose. All the bugs have been reported to the developers. These detected bugs can cause operation failure, data staleness, node downtime, cluster out of service and misleading error message. Among these bugs, the bug zk4283 is a previous known bug, however, the fix of it was not merged to the master and the latest 3.6 branch. The bug hb26420 was not reported before, but it has been fixed in the latest version of HBase 2.4.8.

False positives reported by Deminer are mainly caused by improper checkers, expected failures or failures that can be tolerated by the system. For example, in one of the false positives in HBase, Deminer crashes the meta region server. When the workload is completed, Deminer starts to check the target system. However, at that time, the recovery process for the killed node has not been finished yet, and then the test fails to pass the checker due to non-available meta region. In another false positive from ZooKeeper, the test fails to pass checkers due to a "NullPointerException" thrown during the synchronization process when a follower node follows the leader. However, this exception can be tolerated by restarting the synchronization process with the leader.

5.3 Overhead Analysis

Deminer imposes 1.8x – 5.7x slowdowns in tracing stage, generated 290 to 2108 crash points for each workload, and took 56 to 264 hours for testing generated crashes. The off-line trace analysis is mostly fast and can be finished within one minute. The main factor that contributes to the tracing overhead is the taint propagation. Another key factor is that we instrument the application code at run time. The performance would be better if we statically transform the target systems. The test time can be shortened by configuring the Deminer to reduce the number of max retries and configuring the target system to use a smaller timeout value for crash detection. Given that cloud systems are complicated, the above result demonstrates that Deminer is efficient to be used for real world cloud system testing.

6 RELATED WORK

Fault injection is a commonly used technique for exposing bugs. Chaos Monkey [3] and Jepsen[5] randomly inject node crashes and other types of faults. NEAT [6] provide simple APIs for developers to create and heal network partitions. PreFail [17] allows testers to write failure injection policies to reduce fault injection space. Recent works have also been proposed to use protocol-aware or domain knowledge to conduct fault injections. CORDS [14] systematically injects a single fault to a single file-system block in a single node at a time. CoFI [10] injects network partitions at inconsistent system states. CrashTuner [23] injects node crashes when a node is accessing meta-info variables. These works focus on different fault scenarios from Deminer.

Distributed system model checkers can also be used to expose crash-related bugs. These works intercept non-deterministic distributed events including node crashes and permute their ordering [18, 19, 24, 27, 28]. However, these distributed model checkers do not only focus on crash-related 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.

For bug detection tools for cloud systems, FCatch [22] predicts time-of-fault bugs by observing possible conflicting operations under crashes. ALICE [25] finds crash vulnerabilities of an application by modeling all crash states that can occur on a specific file system. And much previous work has been conducted on detection of other crash-unrelated bugs, including distributed concurrency bugs [21], performance cascading bugs [20], incorrectly-implemented exception handler or missing exception handlers [26, 29], and so on. These works solve an orthogonal problem from Deminer.

7 CONCLUSION

We present Deminer, a novel crash injection approach to test cloud systems. Deminer automatically injects crashes/reboots that interrupt the execution of related operations, by dynamically tracking how data are used at run time and identifying related operation pairs that write common data to different places. Our experiment shows that Deminer has detected six crash recovery bugs on three widely-used cloud systems.

ACKNOWLEDGEMENTS

This work was partially supported by National Natural Science Foundation of China (62072444, 61732019), Frontier Science Project of Chinese Academy of Sciences (QYZDJ-SSW-JSC036), and Youth Innovation Promotion Association at Chinese Academy of Sciences (2018142).

REFERENCES

[1] 2005. *ASM.* https://asm.ow2.io/
[2] 2008. *Apache Hadoop Yarn.* https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html/
[3] 2012. *Chaos Monkey.* https://netflix.github.io/chaosmonkey/
[4] 2016. *Apache Cassandra.* https://cassandra.apache.org/
[5] 2016. *Jepsen.* https://github.com/jepsen-io/jepsen
[6] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *OSDI.* 51–68.
[7] Vaastav Anand. 2018. Dara: Hybrid Model Checking of Distributed Systems. In *ESEC/FSE.* 977–979.
[8] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating Dynamic Data Flow in Commodity Jvms. In *OOPSLA.* 83–101.
[9] Mike Burrows. 2006. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI.* 335–350.
[10] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *ASE.* 536–547.
[11] Jeff Dean. 2009. Designs, Lessons and Advice from Building Large Distributed Systems. Keynote from LADIS.
[12] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI.* 137–149.
[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. In *SOSP.* 205–220.
[14] 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 *FAST.* 149–165.
[15] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *ESEC/FSE.* 539–550.
[16] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *SOSP.* 29–43.
[17] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *OOPSLA.* 171–188.
[18] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. 2007. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI.* 243–256.
[19] 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 *OSDI.* 399–414.
[20] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *EuroSys.* 1–14.
[21] 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 *ASPLOS.* 677–691.
[22] 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 *ASPLOS.* 419–431.
[23] Jie Lu, Chen Liu, Lian Li, Xiaobing Feng, Feng Tan, Jun Ynag, and Liang You. 2019. CrashTuner: Detecting Crash-Recovery Bugs in Cloud Systems via Meta-Info Analysis. In *SOSP.* 114–130.
[24] Jeffrey F Lukman, Huan Ke, Cesar A Stuardo, Riza O Suminto, Daniar H Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S Gunawi. 2019. FlyMC: Highly Scalable Testing of Complex Interleavings in Distributed Systems. In *EuroSys.* 1–16.
[25] 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 *OSDI.* 433–448.
[26] Suman Saha, Jean-Pierre Lozi, Gael Thomas, Julia L. Lawall, and Gilles Muller. 2013. Hector: Detecting Resource-Release Omission Faults in Error-Handling Code for Systems Software. In *DSN.* 1–12.
[27] Jiri Simsa, Randy Bryant, and Garth Gibson. 2010. dBug: Systematic Evaluation of Distributed Systems. In *SSV.* 1–9.
[28] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI.* 213–228.
[29] 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 *OSDI.* 249–265.