FaultFuzz: A Coverage Guided Fault Injection Tool for Distributed Systems

Wenhan Feng[1,2], Qiugen Pei[5], Yu Gao[1,2*], Dong Wang[1,2], Wensheng Dou[1,2,3,4], Jun Wei[1,2,3,4], Zheheng Liang[5], Zhenyue Long[5]
[1]State Key Lab of Computer Science at ISCAS, [2]University of CAS, Beijing, China
[3]Nanjing Institute of Software Technology, [4]University of CAS, Nanjing, China
[5]Joint Laboratory on Cyberspace Security China Southern Power Grid, GuangDong Power Grid, China
{fengwenhan21,gaoyu15,wangdong18,wsdou,wj}@otcaix.iscas.ac.cn
peiqiugen@gd.csg.cn,liangzheheng@xxzx.gd.csg.cn,longzhenyue@gdxx.csg.cn

ABSTRACT

Distributed systems are expected to correctly recover from various faults, e.g., node crash / reboot and network disconnection / reconnection. However, faults that occur under special timing can trigger fault recovery bugs that are rooted in incorrect fault recovery protocols and implementations. Existing random and brute-force fault injection approaches are not effective in revealing fault recovery bugs due to the combinatorial explosion of multiple faults in distributed systems.

In this paper, we propose *FaultFuzz*, a coverage guided fault injection approach that can systematically and effectively test fault recovery behaviors in distributed systems. Based on runtime feedbacks collected from distributed system testing, e.g., code coverage and I/O information, FaultFuzz generates possible combinations of faults, and preferentially selects the combinations that are more likely to trigger new fault recovery behaviors and reveal new fault recovery bugs. We have applied FaultFuzz on three widely-used distributed systems, i.e., Zookeeper, HDFS and HBase and found 5 bugs in them. A video demonstration of FaultFuzz is available at https://youtu.be/SMw1ZF1vyXw.

CCS CONCEPTS

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

KEYWORDS

Distributed system, fault recovery bug, fault injection

ACM Reference Format:
Wenhan Feng, Qiugen Pei, Yu Gao, Dong Wang, Wensheng Dou, Jun Wei, Zheheng Liang, Zhenyue Long. 2024. FaultFuzz: A Coverage Guided Fault Injection Tool for Distributed Systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3639478.3640036

*Yu Gao is the corresponding author. CAS is the abbreviation of Chinese Academy of Sciences. ISCAS is the abbreviation of Institute of Software Chinese Academy of Sciences.

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04
https://doi.org/10.1145/3639478.3640036

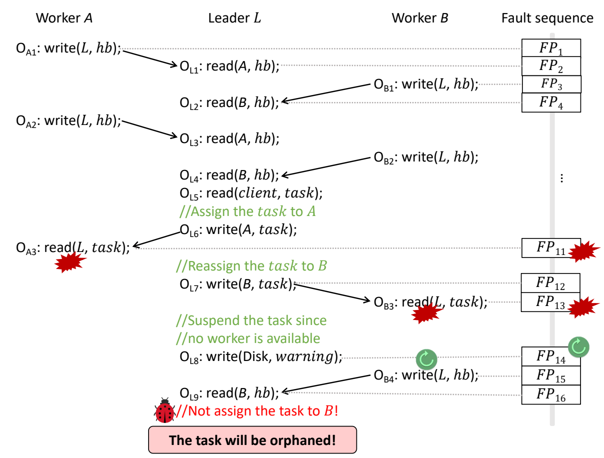

Figure 1: An illustrative fault recovery bug. ***read*** **and** ***write*** **denote I/O operations that read / write data from / to disk or network.** O_{A1} **denotes the first I/O operation on node** A**.** ***hb*** **denotes heartbeat. FP denotes a fault point.**

1 INTRODUCTION

Nowadays, distributed systems [2–4, 6] have been widely-used in many domains, e.g., finance and e-commerce. Large-scale distributed systems usually consist of thousands of nodes that can suffer from various faults at any time [9, 11, 18], e.g., node crash / reboot and network disconnection / reconnection. Distributed systems adopt complex fault recovery protocols to recover from these faults. However, incorrect fault recovery protocols and implementations can introduce fault recovery bugs, and affect the reliability and availability of distributed systems.

We use the illustrative example in Figure 1 to explain how a distributed system handles the faults and how a fault recovery bug is triggered. There are one leader node L, and two worker nodes A and B. Node A and B maintain connections with node L through heartbeat messages, e.g., $O_{A1} \rightarrow O_{L1}$ and $O_{B1} \rightarrow O_{L2}$. When a client submits a task (i.e., O_{L5}), node L assigns the task to node A (i.e., $O_{L6} \rightarrow O_{A3}$). However, node A crashes after receiving the task, thus all in-memory states of A disappear immediately, and the corresponding recovery procedure is triggered. During recovery, node L reassigns the task to node B (i.e., $O_{L7} \rightarrow O_{B3}$). However, B also crashes after receiving the task. In this case, node L suspends the task since no worker is available, and writes a warning into the disk (i.e., O_{L8}). After that, node B reboots. Although node B is available now, node L does not try to reassign the task to B again, and finally makes the task orphaned.

It is challenging to test the correctness of fault recovery in distributed systems through systematically exercising all possible fault scenarios, i.e., all possible combinations of multiple faults. For example, in Figure 1, we can only observe the first 11 I/O points without injecting any faults, i.e., O_{A1} to O_{A3} on node A, O_{L1} to O_{L5} on node L and O_{B1} to O_{B2} on node B. If we inject only one node crash on these 11 I/O points, we can generate 11 fault scenarios. If we inject two node crashes on these 11 I/O points from two different nodes, we can produce (3*6 + 3*2 + 6*2) = 36 fault scenarios. If we further consider injecting more faults (e.g., node reboot) on real-world distributed systems that contain thousands of nodes and I/O points, the number of possible fault scenarios will increase quickly.

Some fault injection approaches have been proposed to detect fault recovery bugs in distributed systems. Random fault injection approaches [5, 7, 24] can miss corner case bugs. Brute-force fault injection approaches [14, 15] exhaustively exercise all possible fault scenarios. Implementation-level model checkers [17, 23, 27] and model-based testing [16, 26] for distributed systems can systematically explore all possible orders of non-deterministic events including faults. They all suffer from the state space explosion problem, and are not effective in exploring the huge fault scenario space in distributed systems. Some approaches only focus on special fault scenarios [10, 13, 20, 22], and cannot be used to systematically test distributed systems. Some distributed system testing approaches [19, 21, 25] cannot be used to explore the fault scenarios.

We observe that some fault scenarios may result in the same recovery behaviors. For example, 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 behaviors, i.e., node L removes the dead node A from its alive node list. This observation inspires us to propose a smarter fault injection approach for distributed systems.

In this paper, we propose a coverage guided fault injection tool *FaultFuzz* based on our previous work [12], which can automatically perform fault injection testing for distributed systems. FaultFuzz represents various fault scenarios as fault sequences, and takes a fault sequence as a special system input to indicate where and when to inject faults. During the system runs, FaultFuzz collects system runtime feedbacks, e.g., I/O and coverage information. Based on these collected information, FaultFuzz generates and mutates new fault sequences. FaultFuzz incorporates an effective fault scenario space exploration strategy to preferentially test the fault sequences that are prone to increase code coverage and trigger new fault recovery bugs. In this way, FaultFuzz can systematically and effectively explore the huge fault scenario space in distributed systems.

We extend our previous work [12] in several aspects, and support more features in FaultFuzz. First, FaultFuzz can support more fault types, i.e., network disconnection and reconnection, and users can flexibly specify their concerned fault types. Second, FaultFuzz can support multiple workloads to drive the test, which can facilitate fault scenario space exploration and bug discovery. Third, FaultFuzz can also support manual annotation of the target distributed system to indicate which application-level I/O points are interesting and should be taken as potential fault injection points. In this way, FaultFuzz can be easily applied to a new distributed system. Finally, FaultFuzz can control more non-determinism among the collected events. Therefore, we can more faithfully reproduce fault sequences during system testing. FaultFuzz has been made publicly available at https://github.com/tcse-iscas/FaultFuzz.

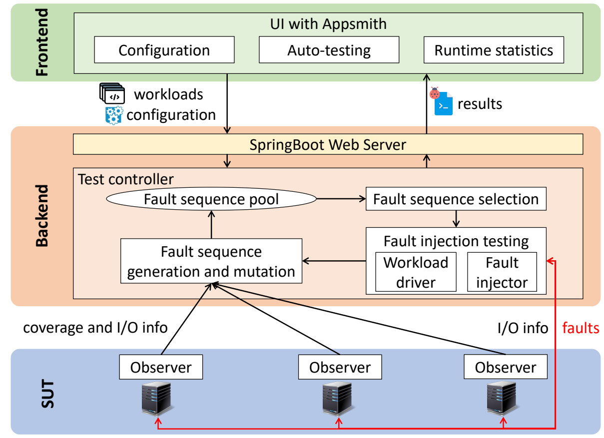

Figure 2: FaultFuzz overview.

2 FAULTFUZZ

Figure 2 shows the overview of FaultFuzz, which contains the frontend, the backend and the system under test (SUT). The frontend provides a web interface and data visualizations for users to configure FaultFuzz, control the testing process, and view runtime test result statistics. The backend is responsible for our core testing logic, including fault sequence generation and mutation, fault sequence selection, and fault injection testing. The SUT is instrumented to collect system runtime information for fault sequence generation and system execution control.

The main testing process contains the following four steps:

- **Information collection:** FaultFuzz collects system runtime feedbacks, e.g., I/O and code coverage information, by instrumenting the target distributed system.
- **Fault sequence generation and mutation:** Based on the collected information, FaultFuzz generates and mutates new fault sequences, and puts them in a pool.
- **Fault sequence selection:** FaultFuzz preferentially selects fault sequences that are prone to increase code coverage and trigger new fault recovery bugs from the fault sequence pool.
- **Fault injection testing:** FaultFuzz utilizes a workload to drive the test, injects faults to SUT according to the selected fault sequence, and uses predefined checkers to detect failure symptoms (e.g., unexpected node downtime) and find bugs.

Without user intervention, FaultFuzz iteratively executes the above four steps until the testing time budget is exhausted or there is no fault sequence to be tested in the fault sequence pool. Note that at the beginning, the fault sequence pool is empty, and we do not have any information that can be used to generate fault sequences. Therefore, FaultFuzz will first run all given workloads separately on SUT without injecting any faults.

2.1 Information Collection

During a system run, FaultFuzz mainly collects two types of information, i.e., coverage and I/O information, by instrumenting the target system through ASM [1].

For coverage information, we use a 64KB byte array to store code coverage information. Each byte in the array corresponds to a basic code block (i.e., a straight-line code sequence with only one entry point and one exit). When a code block is executed, the corresponding byte in the array will be marked as covered.

For I/O information, FaultFuzz can intercept all executed disk and network I/O points at JRE level by instrumenting specific Java APIs, e.g., write APIs in `FileOutputStream` and `SocketOutputStream`. FaultFuzz can also intercept I/O points at application level through automatically instrumenting SUT according to the *@injection* annotations that are specified by developers in SUT code. For example, for ZooKeeper, we can annotate serialize / deserialize APIs in class `Record` which are used for all socket messages in ZooKeeper. We can also directly add calls to the *TriggerAndRecord* function in SUT code. To reduce manual annotation efforts, FaultFuzz has supported automatic instrumentation for Zookeeper, HBase and HDFS to intercept application-level I/O points.

When an I/O point is encountered, FaultFuzz records its corresponding I/O information, including (1) the call stack of an I/O point, (2) the node ID that an I/O point occurs on, (3) the timestamp when an I/O point is executed, (4) destination, which refers to the file path for a disk I/O, or the connected node ID for a network I/O.

2.2 Fault Sequence Generation and Mutation

After a system run, if code coverage is increased, or the last fault injected in this run is node crash or network disconnection (if we inject a corresponding node reboot or network reconnection, the code coverage could increase), FaultFuzz will generate and mutate fault sequences based on the collected runtime information, and add them to the fault sequence pool.

We take I/O points as potential fault injection points, and sort all the I/O points executed in a system run according to their timestamps and obtain a fault sequence. Each fault sequence corresponds to a specific workload that drives SUT. Each I/O point in a fault sequence corresponds to a fault that occurred on the I/O point. FaultFuzz uses fault type (i.e., node crash / reboot and network disconnection / reconnection) and nodes affected by the fault (e.g., the node where the node crash fault occurs) to determine what fault is injected on an I/O point. We leave fault type empty if we do not inject any fault on an I/O point. In Figure 1, we can obtain a fault sequence containing 16 I/O points from three nodes, which corresponds to a workload for submitting a task to leader L. In the sequence, three I/O points O_{A3}, O_{B3} and O_{L8} correspond to a crash on node A, a crash on node B and a reboot on node B, respectively.

For a generated fault sequence *seq*, we mutate it to generate a group of new fault sequences by adding only one feasible fault after the last fault in *seq*. The newly injected fault should satisfy some constraints, e.g., only alive nodes can crash, the number of dead nodes should not exceed the maximum number of dead nodes that the target system can tolerate, etc. Therefore, we can generate valid fault sequences that can be executed by the distributed system.

2.3 Fault Sequence Selection

To effectively explore the fault scenario space in a distributed system, FaultFuzz tries to test fault sequences that are prone to cover new codes and trigger new bugs first. Specifically, FaultFuzz focuses on the last fault $lastFault$ in a fault sequence *seq*. If $lastFault$ is similar to a tested fault, we will test *seq* as late as possible. If $lastFault$ occurs during recovery, we will test *seq* as early as possible. To accelerate the testing process, we also increase testing priorities of the fault sequences that have shorter execution time and larger code coverage. And we try to test fault sequences with multiple faults earlier, as well as avoid testing sequences that contain too many faults, e.g., fault sequences with more than 6 faults.

2.4 Fault Injection Testing

Based on a fault sequence *seq*, FaultFuzz runs the target system with its corresponding workload again, intercepts every concerned I/O point, collects corresponding I/O information, sends the information to the test controller and waits for controller's decision to continue execution or inject a fault. On the controller side, FaultFuzz compares the reported I/O points with the I/O points in *seq*, controls the reported I/O points to be executed in the order of the I/O points in *seq* as much as possible, and injects faults on the reported I/O points according to *seq*. For example, for the fault sequence shown in Figure 1, FaultFuzz will block O_{A3} until all 10 I/O points before it have been executed, and then crashes node A after O_{A3}. If an I/O point in *seq* has not appeared for a long time, we will resume all the blocked I/O points, and only check whether the I/O points for the following faults can be observed. If an I/O point that injects a fault cannot be observed, we will give up this test and put *seq* back to the pool.

3 IMPLEMENTATION AND EVALUATION

We implement the frontend of FaultFuzz using Appsmith [8]. The backend of FaultFuzz includes a web server implemented with SpringBoot and a test controller. The observer of FaultFuzz is a Java agent running on SUT that dynamically instruments the target system through ASM. All the components in FaultFuzz are implemented in around 12,000 lines of Java code in total.

Pause and continue testing. Since the fault scenario space in a distributed system is usually huge and the total testing time can be long, FaultFuzz provides the pause and continue testing capabilities to achieve flexible testing. Before testing a fault sequence, FaultFuzz checks whether it has been paused. If so, FaultFuzz saves the current testing state into disk, e.g., the fault sequences that have been tested and are waiting to be tested, and any intermediate state required to explore the remaining fault scenario space. When FaultFuzz is resumed, it can continue from this state.

Bug reports. FaultFuzz provides detailed bug reports to help developers understand how bugs occur, including the workload and the fault sequence that triggers the bug, the failure symptoms, and the execution logs of the target system and FaultFuzz. With

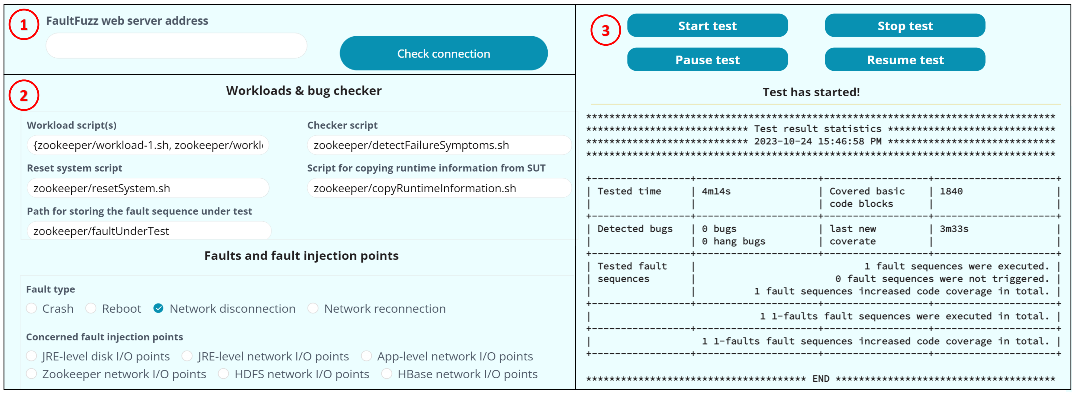

Figure 3: A screenshot of FaultFuzz.

these detailed bug reports, developers can pinpoint the root cause of a bug and figure out how to fix the bug.

Evaluation. We evaluate FaultFuzz on three widely-used distributed systems, i.e., Zookeeper, HDFS and HBase. We use an instrumented JRE for intercepting JRE-level disk I/O points, and use around 332, 990 and 445 lines of code for intercepting application-level network I/O points in Zookeeper, HBase and HDFS, respectively. After running each target system for 48 hours respectively, FaultFuzz has founded 5 bugs in them.

4 DEMONSTRATION SCENARIOS

Figure 3 shows a screenshot of FaultFuzz frontend. Users can use FaultFuzz by the following three steps.

Step 1 (FaultFuzz installation and start). Users first need to install the backend of FaultFuzz, i.e., put the web server jar file and test controller jar file on a host machine. Then users can start the web server through the command "`mvn spring-boot:run`".

We provide a visual frontend as a website on Appsmith cloud. Users can go to the "Check connection" web page, enter the address of the web server, and click the "Check connection" button (Figure 3-①) to confirm that the web server has been started and the frontend can connect to the web server.

Step 2 (Configuration). We provide a "Configuration" web page (Figure 3-②) for users to specify the configurations used to test a target distributed system. The configurations can be divided into four categories, i.e., "Workloads & bug checker", "Faults & fault injection points", "Observer" and "Test controller".

The "Workloads & bug checker" panel allows users to specify the string paths of scripts used for driving SUT and confirming bugs, e.g., the script for requesting SUT, the script for resetting SUT to an initial state, and the script for detecting system failure symptoms. The "Faults & fault injection points" panel allows users to customize concerned fault sequences, such as concerned fault types and fault injection points. The "Observer" panel allows users to specify the information used for instrumenting the target system, e.g., the root path for storing runtime information, the port used by each node in SUT to communicate with the test controller. The "Test controller" panel allows users to specify the information used by FaultFuzz's test controller, e.g., the testing time budget, the path for storing test results, the IP addresses of the nodes in SUT, etc.

After entering the above configuration information, users can click the "Generate configuration files" button to generate and download two configuration files, which should be copied to the server that runs the backend of FaultFuzz and nodes in SUT, respectively. Finally, users need to configure the SUT to use the generated configuration file (and our instrumented JRE if JRE-level I/O points are selected as potential fault injection points) at startup, which will enable dynamic instrumentation for SUT.

Step 3 (Auto-testing and test results). After finishing configuration, users can go to the "Test and result" page (Figure 3-③), enter the path of the test controller jar file and the path of the configuration file. Then users can start automatic fault injection testing for SUT by clicking the "Start test" button. Users can also pause, resume or stop the test by clicking the corresponding buttons.

FaultFuzz displays quantitative statistics of the runtime test results at the bottom of the web page, including the elapsed testing time, the total number of detected bugs, the total number of tested fault sequences, the total number of covered basic code blocks and so on. If the user wants to further observe one specific bug, she can check the corresponding detailed bug report. The user can also try to replay a bug by entering the file path of the fault sequence that triggers the bug and clicking the "Start replay" button.

5 CONCLUSION

We propose FaultFuzz, a coverage guided fault injection tool to systematically and effectively test if a distributed system can recover from various fault scenarios. FaultFuzz leverages runtime feedbacks, e.g., coverage and I/O information, to guide fault scenario generation, mutation and selection. FaultFuzz provides a user-friendly way for developers to test complex real-world distributed systems, and has detected 5 bugs on three widely-used distributed systems.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China (62302493, 62072444, U20A6003), Major Program (JD) of Hubei Province (2023BAA018), Major Project of ISCAS (ISCAS-ZD-202302), China Southern Power Grid Company Limited under Project 037800KK52220005, and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044).

REFERENCES

[1] 2005. *ASM*. https://asm.ow2.io/

[2] 2007. *Apache HBase*. https://hbase.apache.org/

[3] 2008. *Apache Hadoop MapReduce*. https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

[4] 2010. *Apache ZooKeeper*. https://zookeeper.apache.org/

[5] 2012. *Chaos Monkey*. https://netflix.github.io/chaosmonkey

[6] 2015. *Apache Spark*. https://spark.apache.org/

[7] 2016. *Jepsen*. https://github.com/jepsen-io/jepsen

[8] 2019. *Appsmith*. https://www.appsmith.com/

[9] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 51–68.

[10] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. 2020. CoFI: Consistency-Guided Fault Injection for Cloud Systems. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. 536–547.

[11] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-Scale Distributed Systems. In *Proceedings of ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 539–550.

[12] Yu Gao, Wensheng Dou, Dong Wang, Wenhan Feng, Jun Wei, Hua Zhong, and Tao Huang. 2023. Coverage Guided Fault Injection for Cloud Systems. In *Proceedings of International Conference on Software Engineering (ICSE)*. 2211–2223.

[13] Yu Gao, Dong Wang, Qianwang Dai, Wensheng Dou, and Jun Wei. 2022. Common Data Guided Crash Injection for Cloud Systems. In *Proceedings of International Conference on Software Engineering (ICSE Demo)*. 36–40.

[14] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 238–252.

[15] Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. PREFAIL: A Programmable Tool for Multiple-Failure Injection. In *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 171–188.

[16] Beom Heyn Kim, Taesoo Kim, and David Lie. 2022. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models. In *Proceedings of USENIX Annual Technical Conference (ATC)*. 383–398.

[17] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 399–414.

[18] Huang Lexiang, Magnusson Matthew, Muralikrishna Abishek, Bangalore, Estyak Salman, Isaacs Rebecca, Aghayev Abutalib, Zhu Timothy, and Charapko Aleksey. 2022. Metastable Failures in the Wild. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 73–90.

[19] Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: Automatically Detecting Performance Cascading Bugs in Cloud Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–14.

[20] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. 2018. FCatch: Automatically Detecting Time-of-Fault Bugs in Cloud Systems. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 419–431.

[21] Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and Checking Silent Semantic Violations in Large Distributed Systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 91–107.

[22] 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 *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*. 114–130.

[23] 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 *Proceedings of European Conference on Computer Systems (EuroSys)*. 1–16.

[24] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. 2019. Trace Aware Random Testing for Distributed Systems. In *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. 1–29.

[25] Cesar A. Stuardo, Tanakorn Leesatapornwongsa, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, Wei-Chiu Chuang, Shan Lu, and Haryadi S. Gunawi. 2019. Scalecheck: A Single-Machine Approach for Discovering Scalability Bugs in Large Distributed Systems. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*. 359–373.

[26] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. 2023. Model Checking Guided Testing for Distributed Systems. In *Proceedings of European Conference on Computer Systems (EuroSys)*. 127–143.

[27] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 213–228.