What bugs cause production cloud incidents?

Haopeng Liu, Shan Lu
University of Chicago

Madan Musuvathi, Suman Nath
Microsoft Research

ABSTRACT

Cloud services have become the backbone of today's computing world. Runtime *incidents*, which adversely affect the expected service operations, are extremely costly in terms of user impacts and engineering efforts required to resolve them. Hence, such incidents are the target of much research effort. Unfortunately, there is limited understanding about cloud service incidents that actually happen during production runs: what cause them and how they are resolved.

In this work, we carefully study hundreds of high-severity incidents that occurred recently during the production runs of many Microsoft Azure services. We find software bugs to be a major cause behind these incidents, and make interesting observations about the types of software bugs that cause cloud incidents and how these bug-related incidents are resolved, providing motivation and guidance to future research in tackling cloud bugs and improving the cloud-service availability.

CCS CONCEPTS

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

KEYWORDS

Cloud system, production incident, reliability, bug characteristics

1 INTRODUCTION

1.1 Motivations

Cloud services such as distributed computing infrastructures and distributed storage systems have become the backbone of today's computing world. The availability of cloud services is crucial, with minutes of service outages costing millions of dollars [11, 26]. Although much research has attempted to improve the availability of cloud services through automated bug detection [19, 22, 8], failure diagnosis [33], fault detection [16], and others, there is still a lack of understanding about cloud service incidents that actually happen in the wild — what cause them and how they are resolved.

Empirical studies have always been crucial in motivating and guiding the improvement of software availability. Many studies were conducted to understand failure causes and failure resolutions in operating systems [10, 29, 12], multi-threaded software [25], file systems [24], and others [13].

In recent years, empirical studies were also conducted for cloud systems. They mainly use two types of data sources: (1) news reports about cloud outages [15], which contain detailed outage-impact information; (2) open-source bug databases [14, 15, 20, 32, 21], which contain detailed information about bugs found during *both* in-house code review/testing *and* production uses. The focus of these studies has been (1) specific types of bugs (e.g., timing bugs [20], scalability bugs [21], gray component failures [17]); (2) high-level cause categorization (e.g., hardware faults vs. software bugs [14, 15]); and (3) error and failure symptoms (e.g., the scope, propagation, and duration of cloud errors and failures [32, 15, 17]).

Although useful, previous studies *have not* and *cannot*, due to the limitations of their data sources, provide in-depth understanding about *production-run cloud service* incidents, answering fundamental questions like:

(1) What caused production-run service incidents — what types of software bugs escaped in-house testing?

(2) How were production incidents resolved — is there any chance to automate them in the future?

Answers to these questions would be crucial to improving the availability of cloud services.

1.2 Contributions

In this work, we systematically studied **all**[1] the high-severity production-run incidents during a recent span of 6 months in Microsoft Azure services, which cover a wide range of services including computation, storage,

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05…$15.00
https://doi.org/10.1145/3317550.3321438

[1] Except for a few without clear root-cause description (Section 2).

What are the causes of incidents?	
↓	Few hardware problems
↓	Few memory bugs
↓	Few generic semantic bugs
↑	Many fault-detection/handling bugs
↑	Many data-format bugs
↑	More persistent-data races
How are incidents resolved?	
↑	More than half through mitigation w/o patches

Table 1: How are cloud incidents different from failures in single-machine systems? (↑ and ↓ indicate cloud incidents follow certain pattern more or less than single-machine systems.)

data management, data analytics, IoT, media services, etc., and identified software bugs as the most common cause of cloud incidents (close to 40%). We then did an in-depth study of all the 112 high-severity production incidents that are caused by software bugs.

Our study sheds lights on what types of software bugs lead to production cloud incidents, how these incidents are resolved, and how they differ from failures in single-machine systems (Table 1).

What caused incidents? Among all incidents caused by bugs, the most common causes are (1) incorrect or missing detection and handling of component failures (31 %) and (2) inconsistent data-format assumptions held by different software components or versions (21 %). Timing bugs are also common (13 %), with many of them related to conflicting accesses to not only in-memory data but also persistent resources. Finally, incorrectly set constant values are non-negligible (7 %).

Probably related to the rise of bugs related to component failures, we observed the percentage of incidents caused by hardware failures to be significantly smaller (less than 5%) than those in non-cloud systems [13].

How were incidents resolved? Different from bugs in open-source software bug databases, production incidents were more often to get resolved through a mitigation mechanism without a code patch. Those mitigation mechanisms, which we further categorize into code mitigation like rolling back to an older version, running-environment mitigation like killing a process, and data mitigation like deleting a temporary directory, have not been well studied before. Without requiring new code, they provide a good opportunity for incident auto-healing and hence better service availability.

The resolving strategies are different among incidents caused by different types of bugs. For example, running-environment mitigation is the most common resolving strategy for incidents caused by fault-related bugs and timing bugs, but is never used for incidents caused by data-format bugs and constant-value bugs. How to pick the suitable resolving strategy for a production cloud incident is an open problem for future research.

Implications While there has been decades of research on bug detection, in cloud systems, some well studied bugs are much less common (e.g., memory bugs) or are taking new forms (e.g., timing bugs), yet some not-so-well studied bugs (e.g., data-format bugs and fault-related bugs) are taking predominant fractions. Many production-run incidents are resolved through mitigation techniques, instead of patches. Automation techniques that support mitigation would be helpful.

2 METHODOLOGY

2.1 Incidents in our study

Microsoft Azure production incidents can be reported by (Microsoft internal or external) users or by system watchdogs that keep monitoring if certain system metric goes beyond a pre-configured threshold. Every incident is recorded in the incident database, associated with information such as user description or watchdog report, developers' discussion, severity-level tag, root cause description, work items issued to developer teams (if any), the incident-impact duration, etc.

The remainder of the paper focuses on a set of 112 incidents. They are *all* the incidents that satisfy the following four conditions during a 6-month period (March 5th, 2018 – September 5th, 2018):

(1) the incident is not a false alarm and its severity level indicates that new features cannot commit into production environment until this incident is resolved;

(2) the incident led to changes in the cloud service, such as bug-fixing patches, test enhancement, etc;

(3) the incident report contains enough information for us to judge the root cause of the incident;

(4) the root cause of the incident are software bugs.

Note that not all the 112 incidents we studied affected Microsoft 's external customers. Many incidents affected Microsoft 's internal users and many others were detected by internal users and automated watchdogs and mitigated before external customers reported them.

2.2 Threats to validity

The results of our study have to be interpreted with our methodology in mind. The types of bugs we observed in production are biased by the fact that Microsoft uses effective tools (e.g., [4, 5, 6, 7]) to mostly eliminate many types of bugs before they can manifest in production, and hence our study includes zero

or few of such bugs. For example, we observed only a small number of configuration bugs caused by misspecification of configuration entries in configuration files, even though such bugs were reported to be common in other settings[28, 31]). Our observation may not represent incidents in Microsoft that we did not study, and may not represent incidents in other cloud services. We analyzed only incidents whose reports contain enough information for us to judge root causes, which may lead us to miss incidents with complicated root causes and little information about the causes.

3 WHAT ARE THE BUGS?

Every incident report contains a "discussion" section and a "root cause" section, by reading these sections, and sometimes the work item description, we figure out the root cause of each incident, and categorize them into data-format bug incidents (21 %), fault-related bug incidents (31 %), timing bug incidents (13 %), constant-value bug incidents (7 %), and others (28 %).

3.1 Data-format incidents

Different components of cloud services interact with each other through various types of "data", including inter-process/node messages, persistent files, and so on. At the same time, cloud software goes through frequent updates. As a result, different software components in the cloud could hold conflicting assumptions about the format of certain data, leading to service incidents. We refer to these as data-format bugs. They have not been a type of common bugs in traditional software systems [10, 29, 13, 28], but are among the most common ones in our study (21 % of all software bug incidents).

We can categorize these bugs based on the type of data whose format becomes incompatible with newer versions of the software.

(1) Local or global files (about 40% of data-format incidents): different parties assume different formats about certain files or database tables. For example, a service, let's call it Service-X, allows users to store their customized configuration in the cloud. After a feature upgrade, Service-X changes the format of such customized configuration files — a reference to the source configuration has to exist in the customized configuration file. This reference is added to any configuration created by the new-version of Service-X. However, no such reference exists in customized configuration created by earlier versions of Service-X. Therefore, loading old customized configuration files lead to null-reference exceptions and then service incidents.

(2) Message interfaces (about 60% of data-format incidents): a service changes the interface of its external-facing message APIs; consequently, the other process or node that uses this message API got unexpected results. For example, a service, let's call it Service-Y, provides a REST API that returns a list of active instances. In the past, Service-Y used to return an error-code 200 together with an empty list when there were no active instances. In a newer version, Service-Y changed the API to return the error code 404 when there are no active instances, causing incidents in consuming services that could not handle this new return code.

We can also break down these bugs based on different roles of the conflicting parties. They could be caused by inconsistencies between data producers and data consumers (83 % of the cases), as well as between two data consumers (17 % of the cases). In the former case, data produced by one part of the system cannot be properly consumed by another part of the system; in the latter case, different system components draw inconsistent conclusions about whether some user data is valid or not.

In most cases, these bugs are triggered by software updates that fail to fully consider the data-format assumptions held by all stakeholders.

Discussion: Among all the bugs we studied, only one of them occurs inside one process, and the other ones all involve multiple processes and/or nodes. This is probably not a coincidence: persistent data related bugs are more likely to exist in multi-process systems; message related bugs are probably unique to networked systems. The large scale, frequent updates, and long running natures of cloud services likely have facilitated the occurrence of these bugs.

Techniques are needed to automatically extract assumptions about data formats, so that we can automatically detect data-format bugs or automatically raise warnings about inconsistent code versions among different software components.

3.2 Fault-related incidents

Component failures (i.e., faults) are inevitable in cloud environment, and 31 % of software bug incidents are about not detecting or handling faults correctly.

In our study, a component can refer to a user request, a user/system job, a node in the system, a file, and so on. There are three main types of component failures (i.e., faults) that lead to fault-related incidents in our study:

(1) *Error component*: a specific task or job fails and reports an error that cannot be handled by the cloud service platform (43 % among all fault-related incidents);

(2) *Unresponsive component*: a hanging job or a disabled node is not handled by the service platform (i.e., no error code ever returned) and eventually leads to a timeout perceived by users or watchdogs (29 % among all fault-related incidents);

(3) *Silent corruption:* persistent data or cached persistent data became corrupted or inconsistent without any error code and led to incorrect results returned to users (17 % among all fault-related incidents).

We observed three main reasons for a fault in component F not being detected by a component G (in most cases, G then waits infinitely for an operation o, not realizing that o will never occur due the fault in F). 1) G did not contain any fault detection code for potential fault in F; 2) G typically checks certain signal or log to detect faults in F, not realizing that the signal/log itself could disappear due to the fault in F. 3) G typically checks certain signal or log to detect faults in F, not realizing that the signal/log could disappear along the transmission path from F to G. This problem sometimes happens due to file and process re-location after a component failure.

We observed three main types of fault handling problems: (1) handler ignores the error report (35 %); (2) handler over-reacts and causes incidents (35 %); (3) handler contains bugs like infinite loops, timing bugs, etc (30 %). The first two types of problems have also been reported in open-source systems [32].

Discussion: Fault detection and handling is usually not an issue for single-machine systems, but is a major problem in cloud services. This finding is consistent with previous studies about open-source cloud systems [14]. The predominance of fault-related problems confirms our expectation that these bugs only show up in scale and are not likely to be exposed during in-house testing.

Moreover, recent research has looked at various aspects of fault/exception handling problems in distributed systems, including detecting empty error handlers and certain type of over reaction handlers [32], dealing with gray component failures [17, 16], detecting fault-related timing bugs [23], fault injection testing [3], and others. Our study indicates that even with intensive fault-injection testing inside Microsoft (Section 5), fault related bugs are still common, and hence call for more research to help detect and handle faults.

3.3 Timing incidents

Overall, there are 13 % timing incidents in our study set. Among all timing incidents in our study, 72 % incidents are non-deadlock issue and 14 % incidents are deadlock issue. In the remaining 14 % incidents, we only know they are caused by a race condition without any detailed information.

There are two main differences between non-deadlock timing bugs in our study from those in traditional concurrent systems [25].

First, half of these bugs are about race conditions between multiple nodes rather than multiple threads in traditional bugs. Even when a race is among multiple threads, at least one of the threads is an event/message handling thread that is serving the request from a different node like the message-timing bugs discussed in previous empirical studies [20].

Second, half of these bugs are racing on persistent data like cached firewall rules, configuration entries, znodes in Zookeeper, database data, and others, instead of shared memory variables that traditional timing bugs race on. For example, in one case, two system processes read and write the same entry in the machine's configuration file. Races between these two processes' reads and writes led to repeated machine restarts.

Discussion: Timing bugs continue to be a threat to system availability in the cloud. Traditional timing-bug detection techniques need to be adapted to tackle races on persistent data and races between different nodes.

3.4 Constant-value setting incidents

These incidents are caused by an incorrect setting, including typos, of constant variables in the software. They contribute to 7 % of all software bug incidents.

These constant variables include hard-coded configurations, special-purpose strings like URLs, and enum-typed values. For example, cloud software often contains state machines for every node, every job, and so on. The variable that represents the current state of a state machine often has enum type. In some cases, an incorrect constant value of the state variable causes the execution to enter incorrect code path.

Discussion: Comparing with generic and arbitrary typos and semantic bugs, these constant-setting bugs might be easier to automatically discover and fix: some of these bugs are essentially misconfiguration problems; some of these bugs are very easy to fix as there are very few choices for the constant values (considering an enum-typed value).

3.5 Other software bugs

There are about one quarter of the bugs that do not belong to the above four categories. They include 7 resource leak bugs and then 24 miscellaneous semantic bugs. These 7 resource leak incidents include two out-of-memory incidents, four Virtual Machine resource leaks,

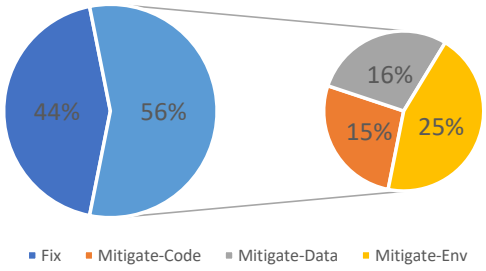

Figure 1: Incident resolve strategy

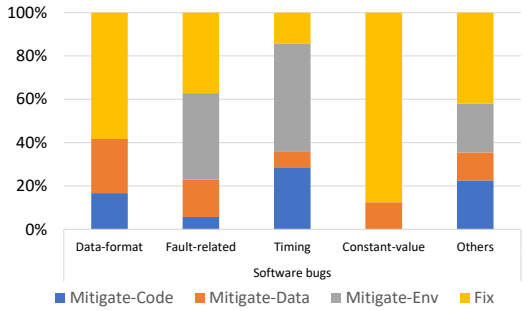

Figure 2: Resolve strategy in each root cause

and one lock leak. Different from that in traditional software systems, memory bugs, other than memory leaks, did not appear at all in our study.

4 HOW WERE THEY RESOLVED?

Facing tight time pressure, more often than not, software-bug incidents were resolved through a variety of *mitigation* techniques (56%) without patching the buggy code (44%), providing quick solutions to users and maximizing service availability. Note that, it is possible that an incident first got resolved by a mitigation technique and later led to a software patch that was not tracked by the incident report.

Q1. *What are the common strategies for mitigating software-bug incidents?*

We categorize all mitigation techniques into three categories: code mitigation, data mitigation, and running-environment mitigation. As shown in Figure 1, these three strategies are all widely used, with environment mitigation the most common in our study.

Code mitigation mainly involves rolling back the software to an older version, or disabling certain code snippets such as an unnecessary/outdated sanity check that failed users' requests and caused severe incidents.

Data mitigation involves manually restoring, cleaning up, or deleting data in a file, a cloud table, etc.

Running-environment mitigation cleans up dynamic environment through killing/restarting processes, migrating workloads, adding fail-over resources, etc.

Q2. *Are different types of bugs resolved differently?*

Figure 2 shows how incidents with different root causes are resolved. As we can see, different types of bugs are indeed resolved differently. Constant-value bugs, and data-related bug incidents are mainly resolved by software patches. On the other hand, environment mitigation is widely used to resolve fault-related bugs, and timing bugs, probably due to the transient nature of many of these incidents and the complexity of handling faults and timing correctly in software.

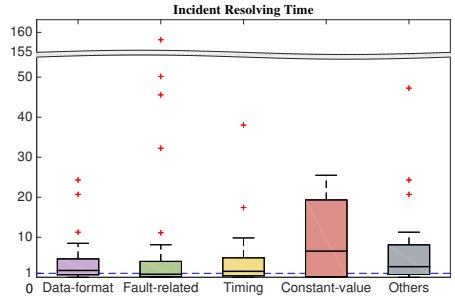

Figure 3: Resolving time for incidents caused by different types of bugs (Y-axis shows the normalized resolving time with the median resolving time of all software-bug incidents as 1; each box represents 25–75 percentile of each type)

Q3. *Do different types of incidents take different amount of time to get resolved?*

Figure 3 compares the normalized resolving time among incidents caused by different types of software bugs, with the median resolving time among all software-bug incidents as "1". As we can see, although the resolving time varies a lot from incident to incident, there is no significant difference among incidents caused by different types of software bugs.

Discussion: Much recent work looked at how to automatically generate new patches. In comparison, automatically generating mitigation steps has not been well studied and worth more attention in the future.

5 PAST AND FUTURE

Many tools have been proposed to detect software bugs, and much focus has been put to avoid bugs during software development. We believe these efforts are reflected in and have influenced the software bug characteristics that we have seen in earlier sections.

The low rate of some bugs is probably related to the tools or languages that are currently used. For example, most of Microsoft Azure is written in .Net managed languages such as C#, and in C/C++, with most C/C++ code inside well-tested legacy components. This is likely the reason that we have seen few memory leak problems and other types of memory problems in our study. Tools like CHESS [27] and PCT [9] are used to expose share-memory concurrency bugs, which contribute to the relatively low rate of those bugs in our study. TLA+ [18, 30] is used to model concurrent and distributed system protocols that allow developers to eliminate high level design/semantic bugs.

At the same time, some types of bugs exist despite the tools and testing already used in house. For example, many Azure services are built on top of Service Fabric [2], which provides Fault Analysis Service [1] that supports various types of fault injections, such as node restart, data migration, random faults, during testing. Although this has been effective in catching fault related problems, the large ratio of fault related bugs indicates that more research is needed.

There are also bugs that have not been tackled by existing tools and deserve future research attention. These include data-format bugs, distributed concurrency bugs on persistent data, and constant-value bugs.

As discussed earlier, much recent research has looked at how to automatically generate patches, a very challenging problem. Our study indicates a likely easier but as important, if not more, direction — how to automatically generate mitigation schemes.

6 RELATED WORK

Given space constraints, we discuss below a few closely related studies on cloud/internet service failures.

A recent paper [15] studied headline news and public post-mortem reports of 597 unplanned outages in 32 different production-run cloud services within a 7 year span. The different data sources led to different focuses and findings in our study and that work. Particularly, accordingly to that study, most (76%) public reports do not discuss details about how outages were resolved, and many (60%) do not explain outage root causes. Consequently, that study focused on outage duration and coarse-granularity cause breakdowns (e.g., upgrade problems versus load problems and so on). Regarding software bugs, it focuses on providing examples of interesting bugs and fixes, yet it cannot and did not answer questions like how common are different types of bugs and resolving strategies.

Yuan et. al. [32] studied 198 user reported failures in 5 open source cloud systems (Cassandra, HBase, etc.). That study intentionally did not look at the root-cause bug types and instead focused on how errors propagate and eventually manifest as failures. Consequently, their study and ours are orthogonal.

Gunawi et. al. [14] studied 3000 issues in the issue system of open source cloud systems, coming from developers' code review, in-house testing, and users' reports (2011 – 2014). They could not check which issues actually caused production incidents and how they were resolved during production (all issues ended up with code patches). Their study found relatively more hardware issues (13%); among software issues, they found less fault-related bugs, although still common (18%), more miscellaneous logic bugs, and did not report data format issues, persistent data timing issues, constant-value issues, and so on. The different observations are likely due to different data sources and study methodology.

There was a study about internet service incidents 15 years ago [28]. Since the authors did not have access to detailed bug reports, their study about incident root causes also stayed at coarse granularity — operator versus hardware versus software. They did not have data about how incidents were resolved during production.

7 CONCLUSION

This paper presented an in-depth study about root causes and resolving strategies of incidents caused by software bugs in production-run cloud services. We hope findings in our study can provide a guidance for future academic and industrial efforts in this field.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful comments. This research is supported by (CCF-1837120, CNS-1764039, 1563956, 1514256, IIS-1546543) and generous support from Microsoft.

REFERENCES

[1] Introduction to the fault analysis service. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-testability-overview, 2017.

[2] Azure service fabric. https://azure.microsoft.com/en-us/services/service-fabric/, 2019.

[3] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Correlated crash vulnerabilities. In *OSDI*, 2016.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *ICSE*, 2019.

[5] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM SIGCOMM*, 2017.

[6] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in microsoft azure. In *International Conference on Distributed Computing and Internet Technology*, 2015.

[7] Ella Bounimova, Patrice Godefroid, and David Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *ICSE*, 2013.

[8] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin T. Vechev. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017.

[9] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM.

[10] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, 2001.

[11] Datapath.io. Recent aws outage and how you could have avoided downtime. https://medium.com/@datapath$_{i}$o/recent-aws-outage-and-how-you-could-have-avoided-downtime-7d9d9443d776, 2017.

[12] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen C. Hunt. Debugging in the (very) large: ten years of implementation and experience. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 2009.

[13] Jim Gray. Why do computers stop and what can be done about it? Tandem Technical report 85.7, 1985.

[14] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[15] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 1–16, New York, NY, USA, 2016. ACM.

[16] Peng Huang, Chuanxiong Guo, Jacob R. Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *13th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '18, pages 1–16, Carlsbad, CA, October 2018. USENIX Association.

[17] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[18] Leslie Lamport. The tla+ home page. http://lamport.azurewebsites.net/tla/tla.html, 2018.

[19] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[20] Tanakorn Leesatapornwongsa, Jeffrey F Lukman, Shan Lu, and Haryadi S Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS*, 2016.

[21] Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability bugs: When 100-node testing is not enough. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[22] Haopeng Liu, Guangpu Li, Jeffrey F Lukman, Jiaxin Li, Shan Lu, Haryadi S Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS*, 2017.

[23] Haopeng Liu, Xu Wang, Guangpu Li, Shan Lu, Feng Ye, and Chen Tian. Fcatch: Automatically detecting time-of-fault bugs in cloud systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 419–431. ACM, 2018.

[24] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux

file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[25] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[26] IHS Markit. Businesses losing $700 billion a year to it downtime, says ihs. http://news.ihsmarkit.com/press-release/technology/businesses-losing-700-billion-year-it-downtime-says-ihs, 2016.

[27] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[28] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *USITS*, 2003.

[29] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in linux: Ten years later. In *ASPLOS*, 2011.

[30] TLA Whence. Leslie lamport: The specification language tla+.

[31] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do not blame users for misconfigurations. In *SOSP*, 2013.

[32] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[33] Yongle Zhang, Serguei Makarov, Xiang Ren, David Lion, and Ding Yuan. Pensieve: Non-intrusive failure reproduction for distributed systems using the event chaining approach. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017.