On the Reproducibility of Bugs in File-System Aware Storage Applications

Duo Zhang[a*], Tabassum Mahmud[a*], Om Rameshwar Gatla[a], RunZhou Han[ab], Yong Chen[b], Mai Zheng[a]
[a] Dept. of Electrical and Computer Engineering, Iowa State University, USA
[b] Samsung Electronics America, USA
{duozhang, tmahmud, ogatla, hanrz, mai}@iastate.edu, {yongchen.1}@samsung.com

***Abstract*—Many storage applications such as file system checkers, defragmentation tools, etc. require a detailed understanding of file systems. Such file-system aware applications play an essential role today, but unfortunately they are error-prone. To better understand the challenges as well as the opportunities to address the issues, this paper presents an empirical study of real world bugs in file-system aware storage applications. By analyzing 59 bug cases from 4 representative applications in depth, we derive multiple insights in terms of general bug patterns, triggering conditions, and implications for building effective tools to address the issues. We hope that our study and the resulting dataset could contribute to the development of reliability tools for building robust file-system aware storage applications in general.**

***Index Terms*—File system, storage system, configuration parameter, reproducibility, reliability**

I. Introduction

Many storage applications require a detailed understanding of file systems (FS). For example, FS checkers (e.g., `e2fsck` for Ext4 [1], `ChkDsk` for NTFS [2]) scan the on-disk layout of file systems, detect metadata inconsistencies, and serve as the last line of defense to bring a corrupted file system back to a healthy state [3], [4]. Similarly, FS resizers [1], defragmentation tools [5], data recovery tools [6], etc. all need to parse file system structures precisely. Such FS-aware storage applications play an essential role in maintaining the storage system reliability and performance today [7].

Unfortunately, developing FS-aware storage applications is challenging due to the complexity of file systems, the lack of documentation or library support, etc [7]. Consequently, such software may contain bugs that cause system failures or even data loss. For example, a recent NTFS `ChkDsk` issue affected many Windows users [8]–[10]. Similarly, `e2fsck` may generate incorrect or insecure repairs, corrupting Ext-family file systems [7], [11]–[14].

Addressing the challenge above requires collective efforts from multiple research directions including bug detection support [12], [13], [15], [16], programming language and library support [7], data provenance and debugging support [17]–[19], etc., all of which will benefit from a better understanding of real-world bug characteristics.

Many studies have been conducted to understand and guide the improvement of storage software [20]–[25]. For example, Lu *et al.* studied 105 bugs from 4 multi-threaded applications (including MySQL database) and found common concurrency bug patterns (e.g., atomicity violation) [20]. Duo *et al.* studied about 310 persistent memory (PM) related bugs in the Linux kernel and identified PM-specific issues (e.g., misaligned NVDIMM namespace) [25]. Such efforts have generated valuable insights for optimizing the corresponding software, but they are not necessarily applicable to FS-aware storage applications due to the different design patterns and tradeoffs.

In this paper, we perform an empirical study on the characteristics of 59 bugs in four FS-aware storage applications including two file system checkers [26], one file system resizer [27], and one degragmentation tool [28]. Different from most existing studies, we take one step further to *reproduce* the bugs in our dataset, which enable us to drive deeper insights along multiple dimensions for developing remedy solutions.

First, in terms of general bug patterns, we find that semantic bugs is the dominant type (98.3%) in our dataset based on the classic taxonomy proposed in the literature [20], [21], [25], which is typically caused by incorrect design logic of the software. Moreover, we observe a few unique subtypes among the *semantic* bugs. For example, 15.3% cases are caused by the inconsistency between the application logic and the file system features (i.e., the application cannot handle certain (new) features of the underlying file systems, which may lead to unexpected abort or even file system corruptions).

Second, in terms of triggering conditions, we find that there are four important factors for reproducing the issues in our dataset, including (1) file system parameters, (2) application parameters, (3) workloads, and (4) metadata corruptions. Specifically, we find that the majority of bugs (96.6%) require certain file system parameters to trigger, 30.5% require application parameters, and 27.1% depend on both. Similarly, 39.0% and 22.0% bug cases depend on workloads and metadata corruptions, respectively. This result reflects the strong dependency between the storage applications and the underlying file systems as well as the complexity for detecting or diagnosing such issues. But interestingly, we find that 19 of the bugs in our dataset could be detected via 2 simple rules.

Finally, based on the findings, we derive multiple implications for building effective tools (e.g., bug detection, debugging, provenance tracking) to mitigate the issues. We curate

*Both authors contributed equally

978-1-6654-5408-7/22/$31.00 ©2022 IEEE

TABLE I
FILE-SYSTEM AWARE STORAGE APPLICATIONS STUDIED IN THIS WORK.

App. ID	Name	Description	# of Bugs	Reproduced Bugs
1FSCK	`e2fsck`	the checking and repair program for Ext-family file systems	37	20 (54.0%)
2FSCK	`xfs_repair`	the repair program for XFS file system	2	0 (0%)
3SIZE	`resize2fs`	the resize program for Ext-family file systems	17	12 (70.6%)
4FRAG	`e2freefrag`	the free space managing program for Ext-family file systems	3	2 (66.7%)
TOTAL	–	–	59	34 (57.6%)

our study results in a software artifact called *FSAppBugs* which includes the detailed characterization of bug patterns as well as the critical conditions and scripts to reproduce them. We have released the artifact publicly to facilitate follow-up research on improving the design and implementation of storage applications and relevant tools[1].

The rest of the paper is organized as follows: §II describes the study methodology; §III presents the study results; §IV discusses the related work; and §V concludes the paper with future work.

II. METHODOLOGY

In this section, we describe how we collect the dataset for study (§II-A), how we analyze the bug characteristics (§II-B), and the limitations of the approach (§II-C).

A. Dataset Collection

As shown in Table I, we look into four FS-aware storage applications including two file system checkers (i.e., `e2fsck` and `xfs-repair`), one file system resizer (i.e., `resize2fs`), and one defragmentation tool (i.e., `e2freefrag`). These applications are developed for maintaining the popular Ext-family (e.g., Ext2/3/4) and XFS file systems on Linux.

All the applications maintain their source code including the commit history in Git repositories, which enables studying the real bugs of the applications that have been confirmed and fixed. Note that not all the committed patches are bug patches. We find that the majority of the patches are for maintenance (e.g., code re-factoring or documentation updates) or adding new functionalities instead of fixing bugs, which is consistent with previous studies on Linux kernel patches [21], [25].

In order to effectively identify potentially reproducible bug cases, we first use a wide set of keywords (e.g., 'bug', 'error', 'corrupt', 'reproduce', 'configuration', 'parameter', 'trigger', 'condition', etc.) to filter the patches, and then manually examine the remaining patches to understand their purposes. At the time of this writing, we are able to identify 59 bug patches which contain relatively complete information (e.g., steps for reproducing the issues) for study.

B. Dataset Analysis

Based on the 59 bug cases, we conduct a comprehensive study to answer three set of questions:

- General Patterns: What are the general bug patterns? Is there anything unique compared to previous studies?
- Triggering Conditions: What are the specific conditions required to trigger and reproduce the bugs? Are there any general rules that can be leveraged to remedy?
- Implications: What are the implications for researchers and practitioners for developing relevant tools (e.g., bug detection, debugging, provenance tracking)?

To answer these questions, we manually analyzed each patch in depth. The patches typically follow a standard format containing a description and code changes, which enables us to infer its logic and characterize them accordingly. For patches that contain limited information, we further investigated relevant source code and design documents. Moreover, we attempt to reproduce the identified bug cases in our experimental environment, which enables us to gain deep insights on the necessary conditions for manifesting the bugs.

C. Limitations

Note that the combination of keyword search and manual examination (§II-A) is similar to the previous studies [20], [21], [25], though most existing studies did not perform the reproducibility experiments. Similar to previous studies [20], [21], [25], our study results should be interpreted with the method in mind. The dataset was refined via keywords and manual examination, and we only studied bugs that have been triggered and fixed by the developers, both of which may lead to incompleteness. Nevertheless, we believe such study is one important step toward better understanding the characteristics of bugs in storage applications and deriving effective solutions. By releasing the study results publicly, we hope the work can inspire follow up research and benefit the community.

III. STUDY RESULTS

A. Overview

As shown in the last column of Table I, we were able to reproduce 34 bug cases (57.6%) successfully in our experimental environment at the time of this writing, including 20 from 1FSCK, 12 from 3SIZE, and 2 from 4FRAG. Reproducing these cases require satisfying various conditions, as will be elaborated in §III-C. For the remaining cases that we cannot reproduce, we derive the necessary (but not sufficient) conditions based on our understanding of the source code and relevant documents. We summarize the findings based on all the cases in this section, including the general patterns (§III-B), triggering conditions (§III-C), and implications (§III-D).

[1] *https://git.ece.iastate.edu/data-storage-lab/prototypes/bugbench/-/tree/main/FSAppBugs*

TABLE II
CLASSIFICATION OF BUG PATTERNS. THE LAST COLUMN SHOWS THE NUMBER OF BUGS (AND PERCENTAGE) BELONGING TO EACH SUBTYPE.

Type	Subtype	Description	# of Bugs (%)
Semantic (98.3%)	Feature	out of sync for FS feature support	9 (15.3%)
	Wrong Check	miss / wrong check of FS corruption	13 (22.0%)
	Wrong Repair	miss / wrong repair of FS corruption	13 (22.0%)
	Logic	other improper design	23 (39.0%)
Memory (1.7%)	Null Pointer	dereference null pointer	1 (1.7%)

TABLE III
MODIFICATIONS INTRODUCED BY BUG PATCHES.

App. ID	Application Code	External Code
1FSCK	37	3 (8.1%)
2FSCK	2	–
3SIZE	17	2 (11.8%)
4FRAG	3	–
TOTAL	59	5 (8.5%)

B. General Patterns

To guide the design of bug detectors and other reliability tools, it is important to understand the types of bug occurred. As summarized in Table II, we find that the 59 bugs in our dataset can be classified into two major types based on the classic taxonomy proposed in the literature [21], [25]: (1) *Semantic*, which means the issues are related to the high-level design semantics of the software; (2) *Memory*, which relates to illegal memory operations (e.g., out-of-bound access).

Similar to previous studies, the *Semantic* type accounts for the majority of bugs in our dataset. This reflects the complexity of developing correct storage applications. On the other hand, we observe a few interesting and unique patterns among the *Semantic* bugs, which we discuss further below:

- *Feature* (15.3%): This type of bugs occurs when the storage applications cannot handle certain features (newly) added to the underlying file systems. In other words, the applications is out-of-sync with the file systems. The consequence is often severe: the application may abort abruptly or even corrupt the underlying file system. This pattern reflects the complex dependencies between the storage applications and file systems and suggests the need for more effective co-designs.
- *Wrong Check* (22.0%) and *Wrong Repair* (22.0%): These patterns are specific to the file system checker bugs in our dataset. We find that FSCK may generate wrong checking results (e.g., missing corruptions on specific metadata structures) or even wrong fix, which is consistent with previous work on the correctness of file system checkers [11]–[13]. More elegant design patterns (e.g., SQL query for checkers [11] or programming lanague support [7]) are likely needed for mitigating the issues.

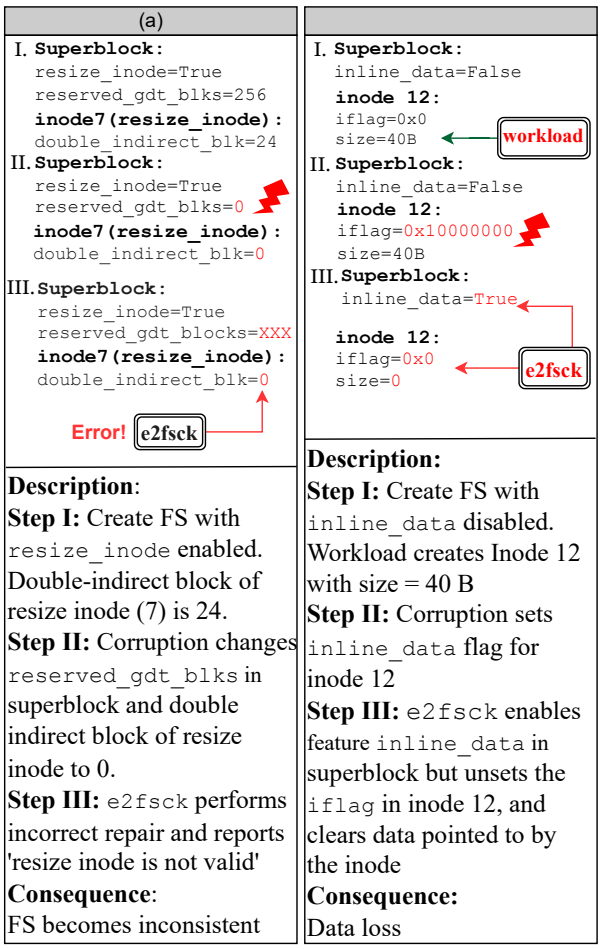

Fig. 1. Examples of Triggering Conditions. This figure shows two specific bug cases which (a) depends on file system parameters and metadata corruptions, (b) depends on workloads and metadata corruptions.

To understand the impact of the bug patches further, we measure the code modifications required by each patch. As summarized in Table III, 5 out of the 59 cases (8.5%) require modifications to external source code (e.g., library or file system code) in addition to modifying their own application code, which reflects the dependency between the applications and their environment.

C. Triggering Conditions

Besides identifying general bug patterns, we further look into specific conditions needed to trigger the bugs, which is critical for developing effective tools to capture similar bugs in practice. We find that four major factors are important for our dataset, including: file system parameters, application parameters, workloads, and metadata corruptions. We discuss them in more details below:

(1) 96.6% bugs depend on file system parameters. The configuration parameters of file systems control the specific features or functionalities of file systems, which may affect the behavior of the upper storage applications. As shown in the first three columns of Table IV, 57 out of the 59 bugs (96.6%)

TABLE IV
DISTRIBUTION OF BUGS BASED ON FOUR MAJOR TRIGGERING CONDITIONS: FILE SYSTEM (FS) PARAMETERS, APPLICATION (APP) PARAMETERS, WORKLOADS, AND METADATA CORRUPTIONS.

App. ID	# of Bugs	Depend on FS Param.	Depend on App. Param.	Depend on Both FS & App Param.	Other Dependency	
					Workload	Corruption
1FSCK	37	35 (94.6%)	5 (13.5%)	3 (8.1%)	19 (51.4%)	13 (35.1%)
2FSCK	2	2 (100%)	–	–	–	–
3SIZE	17	17 (100%)	13 (76.5%)	13 (76.5%)	4 (23.5%)	–
4FRAG	3	3 (100%)	–	–	–	–
TOTAL	59	57 (96.6%)	18 (30.5%)	16 (27.1%)	23 (39.0%)	13 (22.0%)

TABLE V
DISTRIBUTION OF BUGS BASED ON THEIR DEPENDENCY ON SINGLE OR MULTIPLE PARAMETERS

App. ID	Depend on FS params		Depend on App. params	
	single	multiple	single	multiple
1FSCK	24	11	2	3
2FSCK	–	2	–	–
3SIZE	14	3	11	–
4FRAG	3	–	–	–
TOTAL	41	16	13	3

TABLE VI
TWO SIMPLE RULES AND THEIR EFFECTIVENESS.

Rule ID	Description	Bugs Detected
R_1	check return code of target application	4
R_2	compare multiple runs of FSCK	15
TOTAL	–	19

in our dataset require specific file system parameters to trigger, including 35 from 1FSCK (i.e., 94.6% of all 1FSCK bugs), 2 from 2FSCK (100%), 17 from 3SIZE (100%), and 3 from 4FRAG (100%). Most interestingly, we find that in two cases an FS parameter must be *enabled* while another parameter must be *disabled*, which reflects the complex relations in file system configurations.

Figure 1 (a) shows a concrete example where a bug depends on a file system parameter to trigger. In this case, the Ext4 file system has the `resize_inode` parameter enabled (Step I). In case of an unfortunate corruption, the `reserved_gdt_blocks` field in the superblock is changed to zero and the `double_indirect_block` in inode 7 (i.e., resize inode) also becomes zero (Step II). In this scenario, `e2fsck` performs an incorrect repair by clearing the `resize_inode`, and aborts with an error.

Note that a bug may depend on more than one file system parameters. As shown in the second column of Table V, there are 16 cases in total which depend on multiple file system parameters for manifestation.

(2) 30.5% bugs depend on application parameters. As shown in the fourth column of Table IV, 18 bugs in total require specific application parameters to trigger, including 5 from 1FSCK and 13 from 3SIZE. Moreover, Table V further shows that among the 16 cases, 13 of them depend on single application parameter and 3 of them depend on multiple application parameters.

(3) 27.1% bugs depend on both file system and application parameters. Among all the bugs that depend on parameters, 16 are particularly tricky as they cannot be triggered without specific file system parameters and application parameters.

(4) 39.0% bugs depend on workloads. Moreover, we find that 23 bugs require specific workloads to trigger. Figure 1 (b) shows an example from 1FSCK (`e2fsck`). The bug is dependent on the `inline_data` feature as well as a specific workload. In this case, the `inline_data` feature is disabled in the superblock ('False'). The workload creates a file (inode 12) with 40B data ('size=40B'), and the `iflag` (meaning "`inline_data` flag") in the inode is also unset by default (Step I). In case of corruption, the `iflag` may be corrupted ('iflag=0x10000000'). In an attempt to fix the corruption (Step III), `e2fsck` mistakenly enables the `inline_data` in the superblock ('True'), while disabling the `iflag` of the inode ('iflag=0x0') and removing the data from the inode ('size=0'), which leads to data loss.

Such workload-dependent bugs are particularly difficult to trigger because there are numerous ways of using file systems in practice. Therefore, systematic approaches to generate effective workload operations are likely needed for addressing the issue.

(5) 22.0% bugs depend on corruption. As shown in the last column of Table IV, 13 bugs require specific corruption for triggering the bug. In fact, both the examples in Figure 1 fall in this category. In both of the example cases, specific corruptions are needed to trigger the bugs, i.e., corruption in `reserved_gdt_blk` and corruption in `iflag` for Figure 1 (a) and (b), respectively. This is reasonable because file system checkers are designed to scan and fix corrupted images, certain functionalities (and the latent bugs) may only be triggered when processing specific metadata corruption scenarios.

(6) 55.9% reproduced bugs can be detected via 2 general rules. As mentioned in §III-A, we were able to reproduce 34 (57.6%) bug cases successfully in our experimental environment. Most interestingly, we find that among the 34 reproducible cases, 19 of them can be detected via two simple rules once they are triggered.

As summarized in Table VI, the first rule (R_1) means checking the return code of application, which allows us to

detect 4 bugs in total. This is consistent with the previous finding that error checking code is often not implemented properly [29]. The second rule (R_2) means comparing the execution results of multiple runs of the FSCK application, which is useful because FSCK may generate partial fixes [12].

D. Implications

Based on the findings above, we discuss multiple important implications for researchers and practitioners to improve FS-aware storage applications below:

Bug Detection. Great efforts have been made to detect bugs proactively in storage applications [12], [13], [15], [16], [30]. For example, Carreira *et al.*use a mix of symbolic and concrete execution to test three FS checkers (i.e., `e2fsck` [26], `reiserfsck` [31], `fsck.minix` [32]) and find bugs in all of them [12]; Om *et al.*apply fault injections to test the fault resilience of multiple file system checkers [13]; Xu *et al.*apply feedback-driven fuzzing to file systems and expose various issues including 2 bugs in `e2fsck` [30].

Nevertheless, our study reveals that many issues in FS-aware storage applications require specific conditions to manifest (e.g., specific file system parameters or corruptions), which are agnostic to many of the existing bug detectors to the best of our knowledge. In other words, the coverage of many existing bug detectors are fundamentally limited because they do not consider the necessary bug triggering conditions. By demonstrating the general characteristics as well as the reproducibility of bug cases, our study could potentially help improve the coverage and effectiveness of bug detectors for storage applications, which we leave as future work.

Debugging Tools. Once a failure symptom is observed in practice, it is necessary to diagnose the root cause (i.e., the fault/bug that caused the failure), which is notoriously difficult and time-consuming. Many debugging tools have been developed to facilitate the diagnosis procedure [33], [34]. For example, GDB [33] is the *de facto* way to debugging many software failures, which requires substantial manual interactions. Besides GDB, there are many other prototypes proposed to automate the debugging procedure further, including record-and-replay based approaches (e.g., PANDA[34]) and program slicing based approaches (e.g., FailureSketching[17], REPT[35], Giri [36])

Nevertheless, most existing approaches assume that the failure can be reproduced reliably and consistently, which is not necessarily the case in practice. As demonstrated in our study, triggering bugs in FS-aware storage applications may require many conditions, which is essential for debugging the failures caused by such bugs. Our results suggest that constantly monitoring system environments (e.g., file system parameters) and/or tracking data provenance (e.g., lineage of application parameters) might be needed in order to provide sufficient information for reproducing failure scenarios involving FS-aware storage applications.

Provenance Tools. In addition to traditional debuggers, many provenance-based approaches have been proposed for understanding various system behaviors including bug-induced failures [18], [19], [37]–[43]. For example, PASS & PASS v2 [18], [37] collects file system level provenance for understanding the transformations of data in the system transparently. More recently, Hu *et al.* use provenance to help scientists debug their workflows efficiently [19]; Lineage Stash [40] uses provenance in a record & replay manner to handle distributed system failures; Han *et al.* apply provenance-based fault detection in cloud environment [41].

Our study suggests that it is important for provenance tools to capture parameter information of the system or application. Moreover, since FS-aware storage applications often involve multiple programs (e.g., file systems and their utility applications), it is desirable to capture the correlation between different programs in the provenance. This implies that consolidating individual provenances from different programs is likely needed, which we leave as future work.

IV. Related Work

In this section, we discuss related work that has not been covered sufficiently in previous sections, which can be roughly classified into two categories as follows:

Studies of Software Bugs and Storage System Failures. Many researchers have performed empirical studies on the characteristics of bugs in open source software [20]–[25], [44], [45]. For example, Lu *et al.* [20] studied 105 concurrency bugs and found that atomicity-violation and order-violation are the two most common patterns; Chen *et al.* [23] studied 141 Linux kernel bugs and identified their security implications; Thanumalayan *et al.* [46] studied application level crash-consistency protocols and find a total of 60 vulnerabilities, many of which lead to severe consequences; Duo *et al.* [25] studied 1,350 PM-related kernel patches including about 300 bug cases and derived multiple insights in terms of patch characterization, bug patterns, fix strategies, etc. Our study is complementary to the existing efforts as we focus on bugs in important file-system aware storage applications.

One recent work [45] studied the configuration issues in file system ecosystems involving three FS-aware storage applications. Similar to our work, they also recognize the importance of user-level storage applications. Different from our work, they focus on deriving the configuration dependencies for file systems instead of the reproducibility of bug cases. Therefore, these two works are complementary.

In addition to bug studies, researchers have also studied the failures of storage systems [47]–[52]. For example, Xu *et al.* [48] studied SSD-related failures in Alibaba cloud storage. Gunawi *et al.* [47] studied real-world failures of cloud storage services and found that many root causes were not specified well. Han *et al.* [52] analyzed the inconsistent checking and repairing of parallel file systems under failures. In general, these system failure studies are orthogonal our study as they do not investigate the source code level bug patterns or reproducibility of the cases. On the other hand, our study on the bug patterns of storage software and triggering conditions

may help understand the root causes of the failure symptoms observed in the real world.

Reliability Tools for Improving Storage Systems. Many tools have been built to improve the robustness of storage software systems [7], [11]–[13], [51], [53]–[61]. For example, Spiffy [7] creates an annotation language for specifying file system formats and facilitates creating robust storage applications; SQCK [11] proposes an elegant design for file system checkers based on a declarative query language; RFSCK [60] provides a transactional library to strengthen the resilience of file system checkers; PFault [59] injects multiple types of faults to understand the failure handling of parallel file systems.

In general, these tooling efforts are orthogonal to ours which focus on understanding the characteristics of bugs in the storage applications. On the other hand, as elaborated in §III-D, a better understanding of bug characteristics enabled by our work will likely help improve the effectiveness of existing and future tools further.

V. Conclusion & Future Work

We have presented a comprehensive study on the real-world bug cases of representative file-system aware storage applications including two file system checkers, one file system resizer, and one degragmentation tool. We have identified a number of unique bug patterns including the inconsistencies between the application logic and the file system features. Moreover, we have found that four conditions are critically important for the manifestation of the bugs. Based on the identified bug patterns and critical triggering conditions, we derived multiple implications for building relevant tools including bug detectors, debugging tools, and provenance tools.

In the future, we plan to incorporate the insights derived from the study into the development of effective bug detection, debugging, and provenance tools. We hope that this study as well as the resulting artifact could contribute to the evolution of robust storage applications in general.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful feedback. We also thank Wei Xu, Haolun Ping, Carson Love, Ryan Bumann, Jahid Hasan, Kajal Kattige for their help on reproducing a few bug cases and/or validating relevant test suites. This work was supported in part by National Science Foundation (NSF) under grants CNS-1566554/1855565, CCF-1717630/1853714, CCF-1910747 and CNS-1943204. Runzhou Han was supported in part by an internship from Samsung. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] *E2fsprogs: Ext2/3/4 Filesystem Utilities*, http://e2fsprogs.sourceforge.net/.

[2] *chkdsk*, https://www.makeuseof.com/recent-update-windows-10-unexpected-bug/.

[3] A. Ma *et al.*, "Ffsck: The fast file system checker," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[4] D. Domingo *et al.*, "pFSCK: Accelerating File System Checking and Repair for Modern Storage," in *19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021.

[5] *Microsoft Drive Optimizer*, https://en.wikipedia.org/wiki/Microsoft_Drive_Optimizer.

[6] *Data Recovery Softwares*, https://7datarecovery.com/best-recovery-apps/.

[7] K. Sun *et al.*, "Spiffy: Enabling file-system aware storage applications," in *16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[8] *Windows 10 20H2: ChkDsk damages file system on SSDs with Update KB4592438 installed*, https://borncity.com/win/2020/12/18/windows-10-20h2-chkdsk-damages-file-system-on-ssds-with-update-kb4592438-installed/.

[9] *Windows 10 2004/20H2: Microsoft fixes chkdsk issue in update KB4592438*, https://borncity.com/win/2020/12/21/windows-10-2004-20h2-microsoft-fixes-chkdsk-issue-in-update-kb4592438/.

[10] *Microsoft Issues Hotfix For Windows 10 Chkdsk BSODs And SSD File System Corruption*, https://hothardware.com/news/microsoft-issues-fix-for-windows-10-chkdsk-bsods-and-disk-corruption.

[11] H. S. Gunawi *et al.*, "Sqck: A declarative file system checker," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[12] J. C. M. Carreira *et al.*, "Scalable Testing of File System Checkers," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, 2012.

[13] O. R. Gatla *et al.*, "Towards robust file system checkers," in *16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[14] *Recent Update to Windows 10 Introduces Unexpected System Utility Bug*, https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/chkdsk.

[15] *e2fuzz: Create a tool to fuzz ext* filesystems*, https://patchwork.ozlabs.org/project/linux-ext4/patch/20140718225544.31374.24010.stgit@birch.djwong.org/.

[16] W. Xu *et al.*, "Fuzzing File Systems via Two-Dimensional Input Space Exploration," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[17] B. Kasikci *et al.*, "Failure sketching: A technique for automated root cause diagnosis of in-production failures," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP'15)*, 2015.

[18] K.-K. Muniswamy-Reddy *et al.*, "Provenance-aware storage systems," in *Proceedings of the 6th Annual Conference on USENIX Annual Technical Conference (ATC'06)*, 2006.

[19] J. Hu *et al.*, "Improving data scientist efficiency with provenance," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*, 2020.

[20] S. Lu *et al.*, "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'08)*, 2008.

[21] L. Lu *et al.*, "A Study of Linux File System Evolution," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[22] A. Chou *et al.*, "An Empirical Study of Operating Systems Errors," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, 2001.

[23] H. Chen *et al.*, "Linux Kernel Vulnerabilities: State-of-the-art Defenses and Open Problems," in *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys'11)*, 2011.

[24] H. S. Gunawi *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 (SoCC'14)*, 2014.

[25] D. Zhang *et al.*, "A study of persistent memory bugs in the linux kernel," in *Proceedings of the 14th ACM International Systems and Storage Conference (SYSTOR'21)*, 2021.

[26] *e2fsck: check ext2/3/4 file system*, https://linux.die.net/man/8/e2fsck.

[27] *resize2fs: ext2/ext3/ext4 file system resizer*, https://linux.die.net/man/8/resize2fs.

[28] *e2freefrag: report free space fragmentation information*, https://linux.die.net/man/8/e2freefrag.

[29] D. Yuan *et al.*, "Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[30] S. Kim *et al.*, "Finding Semantic Bugs in File Systems with an Extensible Fuzzing Framework," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*, 2019.

[31] *Reiserfsck*, http://manpages.ubuntu.com/manpages/xenial/man8/reiserfsck.8.html/.

[32] *fsck.minix: check consistency of Minix filesystem*, https://man7.org/linux/man-pages/man8/fsck.minix.8.html.

[33] *GDB: The GNU Project Debugger*, https://www.gnu.org/software/gdb/.

[34] B. Dolan-Gavitt *et al.*, "Repeatable reverse engineering with panda," in *5th Program Protection and Reverse Engineering Workshop*, 2015.

[35] W. Cui *et al.*, "Rept: Reverse debugging of failures in deployed software," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, 2018.

[36] S. K. Sahoo *et al.*, "Using Likely Invariants for Automated Software Fault Localization," in *n Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 2013.

[37] K.-K. Muniswamy-Reddy *et al.*, "Layering in Provenance Systems," in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference (ATC'09)*, 2009.

[38] Y. Wu *et al.*, "Diagnosing missing events in distributed systems with negative provenance," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 383-394, 2014. DOI: 10.1145/2740070.2626335.

[39] A. Tabiban *et al.*, "ProvTalk: Towards Interpretable Multi-level Provenance Analysis in Networking Functions Virtualization (NFV)," in *The Network and Distributed System Security Symposium 2022 (NDSS'22)*, 2022.

[40] S. Wang *et al.*, "Lineage stash: fault tolerance off the critical path," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles(SOSP'19)*, 2019.

[41] X. Han *et al.*, "FRAPpuccino: Fault-detection through Runtime Analysis of Provenance," in *9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'17)*, 2017.

[42] T. Pasquier *et al.*, "Practical Whole-System Provenance Capture," in *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*, 2017.

[43] R. Han *et al.*, "PROV-IO: An I/O-Centric Provenance Framework for Scientific Data on HPC Systems," in *The 31st International Symposium on High-Performance Parallel and Distributed Computing (HPDC'22)*, 2022.

[44] D. Lazar *et al.*, "Why does Cryptographic Software Fail? A Case Study and Open Problems," in *Proceedings of 5th Asia-Pacific Workshop on Systems (APSys'14)*, 2014.

[45] T. Mahmud *et al.*, "Understanding Configuration Dependencies of File Systems," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, 2022.

[46] T. S. Pillai *et al.*, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[47] H. S. Gunawi *et al.*, "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages," in *Proceedings of the 7th ACM Symposium on Cloud Computing (SoCC)*, 2016.

[48] E. Xu *et al.*, "Lessons and Actions: What we Learned from 10K SSD-related Storage System Failures," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[49] H. S. Gunawi *et al.*, "Fail-slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems," in *ACM Transactions on Storage (TOS)*, 2018.

[50] E. Xu *et al.*, "Understanding SSD Reliability in Large-scale Cloud Systems," in *Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW)*, 2018.

[51] M. Zheng *et al.*, "Reliability Analysis of SSDs under Power Fault," in *ACM Transactions on Computer Systems (TOCS)*, 2017. [Online]. Available: http://dx.doi.org/10.1145/2992782.

[52] R. Han *et al.*, "Fingerprinting the checker policies of parallel file systems," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW)*, 2020, pp. 46–51. DOI: 10.1109/PDSW51947.2020.00013.

[53] J. Cao *et al.*, "PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems," in *Proceedings of the 2018 International Conference on Supercomputing (ICS)*, 2018, pp. 1–11. DOI: 10.1145/3205289.3205302.

[54] D. Zhang *et al.*, "Benchmarking for Observability: The Case of Diagnosing Storage Failures," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations (TBench)*, vol. 1, no. 1, 2021.

[55] D. Zhang *et al.*, "SentiLog: Anomaly detecting on parallel file systems via log-based sentiment analysis," in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'21)*, 2021, pp. 86–93.

[56] M. Zheng *et al.*, "Understanding the robustness of SSDs under power fault," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, 2013.

[57] M. Zheng *et al.*, "Torturing Databases for Fun and Profit," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 449–464, ISBN: 978-1-931971-16-4.

[58] O. R. Gatla *et al.*, "Understanding the fault resilience of file system checkers," in *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'17)*, 2017. DOI: 10.5555/3154601.3154608.

[59] R. Han *et al.*, "A Study of Failure Recovery and Logging of High-Performance Parallel File Systems," *ACM Transactions on Storage (TOS)*, vol. 18, no. 2, 2022, ISSN: 1553-3077. DOI: 10.1145/3483447. [Online]. Available: https://doi.org/10.1145/3483447.

[60] O. R. Gatla *et al.*, "Towards robust file system checkers," *ACM Transactions on Storage (TOS)*, vol. 14, no. 4, 2018. DOI: 10.1145/3281031.

[61] J. Cao *et al.*, "A generic framework for testing parallel file systems," in *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, 2016. DOI: 10.1109/PDSW-DISCS.2016.013.