Understanding Security Vulnerabilities in File Systems

Miao Cai[*]
Nanjing University

Hao Huang
Nanjing University

Jian Huang
UIUC

ABSTRACT

File systems have been developed for decades with the security-critical foundation provided by operating systems. However, they are still vulnerable to malware attacks and software defects. In this paper, we undertake the first attempt to systematically understand the security vulnerabilities in various file systems. We conduct an empirical study of 157 real cases reported in Common Vulnerabilities and Exposures (CVE). We characterize the file system vulnerabilities in different dimensions that include the common vulnerabilities leveraged by adversaries to initiate their attacks, their exploitation procedures, root causes, consequences, and mitigation approaches. We believe the insights derived from this study have broad implications related to the further enhancement of the security aspect of file systems, and the associated vulnerability detection tools.

1 INTRODUCTION

File system (*fs*) is crucial to the data integrity and security in modern computer systems. Although it has been developed for decades with applying numerous data protection techniques such as access control and sanity checking [6, 17, 20, 24, 25, 32], file systems are suffering from a significant number of malware attacks, and they often fail to protect users from severe damages, such as data loss and leakage, denial of service (DoS), systems crashes, and even full system compromise [15, 34, 36, 37].

Although prior research has performed intensive studies on file systems bugs [16, 21, 29], bug-detection tools [27, 41–43], and formal verification for bug-free implementation [1, 3, 5, 33], few studies investigate their security vulnerabilities. There is no quantitative research demonstrating their root causes and consequences. Moreover, there is no study demonstrating how malicious users exploit these vulnerabilities to successfully initiate the attacks, and pose a great threat to the user data and even the safety of the whole system.

In this paper, we conduct the first systematic study on the security vulnerabilities in Linux file systems. We study 157 real-world cases related to file systems from the list of Common Vulnerabilities and Exposures (CVE). These cases are committed from the year of 1999 to 2019, and cover a variety of file systems that include eight on-disk *fs* implementations such as Ext4, XFS [38], and F2FS [18], two in-memory *fs* like tmpfs, and the virtual file system (VFS). Note that our study mainly focuses on the security aspects of the *fs* design and implementation. We use the CVE list as our resource pool, because all the reported cases are confirmed by security experts, and they represent the real-world threat models.

In order to fully understand each security case, we develop a vulnerability analysis model, which includes three major steps: vulnerability reproducing, attack exploitation, and consequence confirmation. We use this analysis model to guide our study of all the cases, and investigate the *fs* vulnerabilities in different dimensions, including what are their major consequences? what are the common root causes of these vulnerabilities? what are the popular *fs* components that have been exploited by attackers? and how attackers leverage the *fs* features to initiate their attacks?

To be specific, we identify four major types of consequences that include denial of service (DoS) (75%), data leakage (12%), access permission bypass (7%), and privilege escalation (6%), in which the DoS is the major consequence of *fs* vulnerabilities. As for the root causes of these vulnerabilities, we find that they are mainly caused by sanity checking (45%), memory errors with *fs* data structures (23%), race condition in concurrency implementation (8%), and file permission (10%). These are the common issues that have been exploited by attackers. Unfortunately, it is challenging to automatically detect and fix many of them, especially for those that are closely related to high-level system semantic and specifications [13], such as sanity checking issues.

Furthermore, as we map the *fs* vulnerabilities to *fs* core components, including namespace management, *inode* management, *superblock* management, block allocation, page cache, file management, crash-safety model, permission model,

[*]The work was performed when the author was a visiting student in the Systems and Platform Research Group at UIUC.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APSys '19, August 19–20, 2019, Hangzhou, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6893-3/19/08…$15.00
https://doi.org/10.1145/3343737.3343753

Table 1: Summary of Linux *fs* vulnerabilities.

Name	Description	Release Year	#CVEs
Ext4	A journaling fs with extents	2008	39
VFS	Virtual file system	1995	37
XFS	A journaling fs created by SGI	1993	20
F2FS	A flash-friendly fs	2013	15
Btrfs	A copy-on-write based fs	2009	13
procfs	A virtual memory file system	2001	9
Ext2	An extended fs	1993	6
Ext3	A journaling fs	2001	6
tmpfs	A virtual memory fs	2001	5
ReiserFS	A journaling fs created by Namesys	2001	4
JFS	A journaling fs created by IBM	1990	3
Total			157

and *dentry* management, we pinpoint that metadata management is the most vulnerable component in file systems, which occupies 74% of the total *fs* vulnerabilities. As we expected, *fs* features could facilitate attackers to compromise the file systems and even the entire computer systems. We validate that adversaries usually exploit the unique *fs* features such as block management and data consistency model to increase the chances of successful attacks. For instance, attackers would exploit the data inconsistency between page cache and disk with hole punching operations to cause disk data corruption in Ext4 (see `CVE-2015-8839`). We wish our findings could facilitate the file systems development on the aspects of systems security and data protection, as well as the associated vulnerability detection tools.

The rest of this paper is organized as follows. § 2 describes our study methodology. § 3 presents the consequences and causes. We discuss how *fs* vulnerabilities are related to *fs* components in § 4, and present how attackers would exploit *fs* features to initiate their attacks in § 5. § 6 presents the related work. We concludes the paper in § 7.

2 STUDY METHODOLOGY

In this work, we focus on the study of the security vulnerabilities in widely used file systems. We collected the cases related to the file systems by searching the keywords of "file systems" and *fs* names like "Ext4" via the search functionality of CVE. We use the CVE cases as our study samples for two major reasons. First, the published vulnerabilities through the CVE have been confirmed by independent experts [30], they are real cases that provide insightful information. Second, each vulnerability could be confirmed or acknowledged by multiple vendors or platforms, which reflects its impact on different computing platforms.

Study samples. In our study, we mainly examine the cases that have a comprehensive and detailed description in the vulnerability database. Therefore, we study 157 CVE cases in total (see Table 1). These examined cases are reported over the last twenty years from 1999 to 2019, which covers a large portion of the vulnerabilities in many classical file systems such as Ext4, Ext2, and Btrfs that are still in use today. In order to precisely categorize each CVE case, we manually examine the committed report, problem description, and posted blogs following the approaches described in prior bug-study work [11, 21]. For the cases whose source codes are available, we also check the corresponding source codes and the associated committed patches, and reproduce them to further understand and confirm the vulnerability.

Similar to prior characteristic studies that may suffer from limitations on sampling, we take our best effort to collect the vulnerabilities available in the CVE list. Given that we focus on the popular and representative file systems, we believe that these limitations do not invalidate our study result. Also, we encourage readers to focus on the root causes behind each individual case rather than the precise numbers, since a single vulnerability could produce massive damages.

Vulnerability analysis model. To facilitate our study, we develop a vulnerability analysis model, which includes three major steps: vulnerability reproducing, attack exploitation, and consequence confirmation, as described as follows.

- **Vulnerability reproducing.** To initiate a successful attack, the adversary has to verify the effectiveness of the vulnerability, as it reveals the weakness of the file systems. Typical file system weaknesses include poor isolation between namespaces, and insufficient enforcement of file permission model. In this step, we reproduce the vulnerabilities according to the external references in the CVE. For each vulnerability, we generate a reproduction report which describes the conditions to trigger the specific vulnerability.
- **Attack exploitation.** Adversary would combine various attack methods to conduct the attack. Typical attack methods include heap spray, return oriented programming (ROP), and buffer overflow. Similar to the vulnerability reproduction, we also generate a report for the attack exploitation, which records the detailed attack methods of the exploitation and adversary capability.
- **Consequence confirmation.** With attack exploitation, the adversary would initiate the attacks to the file systems and even the entire computer system. Typical consequences include denial of service, data leakage and loss, access permission bypass, and privilege escalation. We confirm the consequences with the damages reported in the CVE.

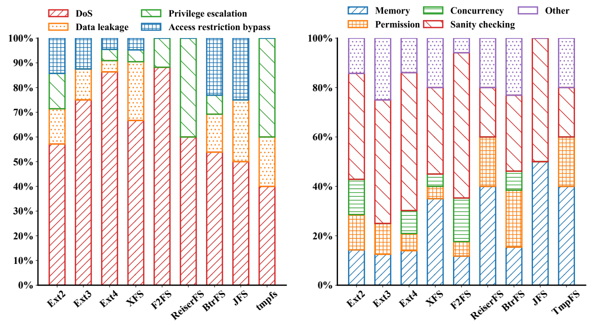
3 CONSEQUENCE AND CAUSES

In this section, we summarize the major consequences and causes of the studied *fs* vulnerabilities.

3.1 Consequences of FS Vulnerability

It is not uncommon that adversaries exploit *fs* vulnerabilities to cause denial of services, steal or destroy user data, and obtain kernel privilege to execute further attacks. We demonstrate the distribution of the common consequences in different file systems in Figure 1a.

RIGHTSLINK Copyright Clearance Center



(a) Consequences **(b)** Causes

Figure 1: Consequences and causes of *fs* vulnerabilities.

Denial of service (75%): we find that the Denial of Service (DoS) is the dominant consequence of *fs* vulnerabilities. Specifically, we identify four major types of DoS caused by *fs* vulnerabilities: kernel crash (35%), memory corruption (16%), memory consumption (13%), and system hang (9%).

- *Kernel crash* and *memory corruption*: file systems are managing complicated storage states such as *inode* in the host memory, it is inevitable that malicious users can exploit the memory errors (e.g., NULL/invalid pointer deference and out-of-bounds memory access) to generate memory corruptions and kernel crashes.
- *Memory consumption*: it is another way to achieve DoS by exhausting the physical memory of the system. File systems request memory from kernel space via *slab*-based memory allocator. If the allocated memory objects are not freed properly, malicious attackers can exploit such types of memory leakage to quickly consume the memory space. For example, XFS does not free the kernel heap memory in its file extended attribute module (`CVE-2016-9685`), attackers can exploit this vulnerability by keeping issuing relevant *fs* operations, which leads to the DoS eventually.

```
static int __get_data_block(struct inode *inode, ...) {
  if (!err) {
    ...
- bh->b_size = map.m_len << inode->i_blkbits;
+ bh->b_size = (u64)map.m_len << inode->i_blkbits;
  }}
```

Listing 1: Attackers exploit an infinite loop in allocating data blocks in F2FS to achieve DoS attack.

- *System hang*: according to our study, it is mainly caused by the over-consumption of CPU resource. We find that the *infinite loop* and *deadlock* are the two main causes to the system hang in file systems. For instance, as discussed in List 1 (`CVE-2017-18257`), attackers could exploit the integer overflow to trigger an infinite loop in data block allocation in F2FS. The type conversion from an unsigned integer to a `size_t` results in an integer overflow, as a consequence, the *bh→b_size* becomes zero. Therefore, the function `f2fs_fiemap` invoking the `get_data_block` will run into an infinite loop.

Table 2: Causes of Linux file system vulnerabilities.

Causes	Sub-type	Description
Sanity checking	miss check	Check is missing
	inapt check	Inappropriate check
Memory error	null ptr	Null pointer deference
	invalid ptr	invalid pointer deference
	integer	Integer overflow/underflow
	uninit data	Uninitialized data structure
Concurrency	race condition	Task synchronization
	lock order	Incorrect lock order
File permission	miss perm	Missed permission check
	inapt perm	Inapt permission check
Other	N/A	Other causes

Data leakage (12%): we find that the data leakage happened in file systems mainly through three ways: (1) the critical data (e.g., kernel stack/heap memory, file cache, and block data, see `CVE-2011-0711`) could be copied from the file system to the user space. Once the data is exposed to the untrusted world, attackers can use these crucial kernel information to perform other severe attacks, such as *code reuse attack*. (2) attackers could leverage race condition occurred in the OS kernel to bypass the file permissions to access the sensitive file content (e.g., `CVE-2012-4508`). (3) attackers could exploit the uninitialized data structures in file systems to obtain critical system information, which includes the file data, journal descriptor block, and kernel memory (e.g., `CVE-2005-0400`, `CVE-2004-0177`).

Access permission bypass (7%): file systems use the permission model to restrict user access behavior on files. When users access a file, they should obey the restriction in the traditional file permission model and access control list (ACL). As a result, if a file system misses or performs an insufficient checking for file permissions, attackers would bypass the permission control of file accesses, therefore, critical files could be read or modified.

Privilege escalation (6%): it allows users with a lower privilege to access data or perform certain operations reserved for the higher privilege. With higher privilege gained through file system vulnerabilities, adversaries can exploit further attacks against the whole system. For instance, as shown in `CVE-2010-1146`, a vulnerability in ReiserFS enables unprivileged users to access the `.reiserfs_priv` directory that stores the attributes (i.e., `xattrs`) of files. Henceforth, malicious users could modify the attribute of any file including the POSIX ACLs, which compromises the access control of the entire file system.

Summary: *By exploiting the vulnerabilities in file systems, adversaries can execute various attacks against the system, including DoS, data leakage, access permission bypass, and privilege escalation. Among those attacks, the DoS is the dominant threat, which is more prevalent than data leakage.*

```
STATIC int xfs_ioc_fsgetxattr(xfs_inode_t *ip,int attr,...){
  struct fsxattr fa;
+ memset(&fa, 0, sizeof(struct fsxattr));

  xfs_ilock(ip, XFS_ILOCK_SHARED);
  fa.fsx_xflags = xfs_ip2xflags(ip);
  if (copy_to_user(arg, &fa, sizeof(fa)))
    return -EFAULT; }
```

Listing 2: Uninitialized `fsxattr` data structure.

3.2 Causes of FS Vulnerability

We carefully explore the root causes of the Linux *fs* vulnerabilities and summarize them in Table 2 and Figure 1b.

Sanity checking (45%): We find that *sanity checking* is a major cause of the *fs* vulnerabilities. However, a majority of them are related to the *fs* semantics. These sanity checks focus on validating *fs* states, such as *fs* namespace, *inode* attributes, file path length, file category (i.e., *regular file*, *symbolic link*), and file permissions.

It is challenging to detect and fix these vulnerabilities since they are restricted by the high-level system semantics and specifications. Recently, researchers are leveraging formal method to verify the correctness of file system implementations [2, 3, 5, 33]. However, as the *fs* codebase is increased dramatically (e.g., the codebase of Ext4 has increased by 3× since the Linux kernel 2.6.19), it requires significant efforts to achieve a full system verification.

Memory errors (23%): File systems are utilizing various in-memory data structures to improve their performance. We observe that a significant number of vulnerabilities are caused by the *uncleaned data structures* in file systems. As discussed in List 2, XFS does not initialize the file extended attribute variable `fsxattr` allocated from kernel stack memory. However, `fsxattr` would be copied to user space in line 9, resulting in the leakage of sensitive kernel information.

As file systems cache substantial disk-related data in memory, the state entanglement between memory and disk causes many security vulnerabilities. For instance, XFS will write in-memory data (including the data structures that are not properly initialized) to the underlying storage device, malicious users can obtain sensitive information by reading data from the raw storage device (`CVE-2004-0177`).

Concurrency (8%): Adversaries usually exploit the race conditions in the concurrency implementation to initiate their attacks. As multiple tasks execute concurrently, the race condition could generate an uncertain period of time, during which malicious users can update the values of the shared data structures. For example, system crash could happen when an asynchronous I/O request and the `fcntl` system call are executed concurrently as demonstrated in Figure 2. The thread `T1` issue a `write` request to Ext4 in step s1. Since this file does not set the `O_DITECT` flag, Ext4 will not initialize

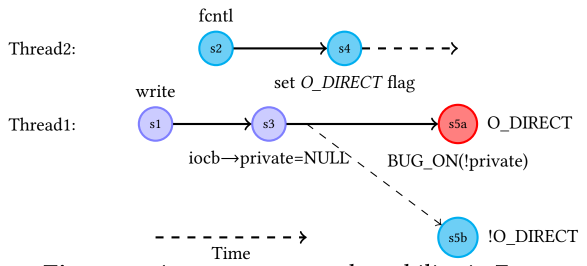


Figure 2: A concurrency vulnerability in Ext4.

the *iocb→private* variable in step s3. Suppose another thread `T2` issues a `fcntl` system call, it would set the `O_DIRECT` flag of the shared file. This will change the execution path of the thread `T1` from `step s5b` to `step s5a`. Finally, the kernel would hit a *BUG_ON* once it performs sanity checking on `iocb→private` variable, resulting in kernel crash.

File permissions (10%): The basic permissions (`r/w/x`) of a file are preserved in the *file* and *inode* structure. Besides that, file systems also support fine-grained, flexible, and powerful access control list (ACL) for users. We find that most vulnerabilities happen because of missing or insufficient checking of these permissions. Adversaries could leverage these vulnerabilities to bypass the access restriction, leading to privilege escalation.

Summary: *The file system vulnerabilities are mainly caused by the sanity checking and memory errors such as uncleaned data structures. However, most of their causes are related to system semantics, which are not easy to be detected with the existing bug-finding tools.*

4 VULNERABLE FS COMPONENTS

We list the vulnerable *fs* components in Table 3, according to our study on the vulnerability distribution in file systems. We will discuss these vulnerable *fs* components respectively.

4.1 Inode Management

Our study shows that the core *fs* component – inode management is the most vulnerable in file systems, in which the extended attribute management and file inline data management are the worst parts (74%). This is probably because of its flexibility and complexity. For example, Ext4 supports three on-disk formats to store extended attributes, including inside *inode*, additional block, and dedicated *inode*. When the size of file extended attribute is small, Ext4 stores the extended attributes inside the *inode* by default. When the attribute size is large, Ext4 will use an additional block to store the extended attributes. Vulnerabilities would happen when we adjust the store format of these extended attributes (e.g., `CVE-2018-11412`). Similar issues will happen in the inline data management, adversaries would exploit these vulnerabilities to crash the file systems.

RIGHTSLINK

Table 3: The most vulnerable *fs* components in Linux.

Name	#CVEs
Inode management	43
Permission model	22
Page cache	13
Block allocation	13
Superblock management	12
File management	11
Crash-safety model	7
Dentry management	6

4.2 File Permission Model

File system permission model restricts the file access for users. Currently, it has two parts: (1) traditional file permission, i.e., `u/g/o:r/w/x`; (2) POSIX access control list (ACL). Almost all *fs*-related system calls require correct file access restriction. As confirmed in § 3, we find that most vulnerabilities happened in the permission model component are caused by missing or insufficient permission checks. Adversaries could exploit these vulnerabilities to bypass the file access permission and thus gain unauthorized kernel privileges. Although this type of security bug is trivial, we find that a large number of *fs* calls suffer from this security issue.

Most importantly, we find that the current file systems do not have rigorous and formal specification for the permission model. System developers have to manually perform permission checks on the system-call execution path to enforce the permission model. Inevitably, developers could miss or perform insufficient permission checking.

4.3 Page Cache

Modern file systems use page cache to improve application performance. Specifically, page cache usually stores metadata blocks temporarily. File systems would flush these metadata blocks to the underlying device via *fsync*-based system calls. However, if system developers forget to initialize certain fields of metadata structure in the page cache, these uninitialized data structures, which contain the kernel sensitive information, would be persisted to the underlying storage device. As a result, a malicious attacker could obtain such sensitive data via reading the raw device, and thus leverage such kernel information to perform further attacks (e.g., `CVE-2005-0400`).

4.4 File Block Organization

File system adopts various strategies to organize the file blocks. Previous file system like Ext2/Ext3 use *direct/indirect* block to organize the file block. Currently, popular file systems (e.g., Ext4, BtrFS) prefer to use *extent* to manage file blocks. Although extent could manage file blocks efficiently, however, our study shows that extent management is much more error-prone than direct/indirect block organization. Their causes are diverse, including improper extent operation, race condition, and integer overflow. Ext4 supports both extent-based and direct/indirect block organization. As system developers take great effort to develop extent management, the size of its codebase (7K lines of codes) is much larger than the codebase of direct/indirect block organization (1.5K LoC). Such complexity obviously generates more security issues (9×) than direct/indirect block organization. As more features are integrated into file systems, such as flexible/metadata block group and delay block allocation, they bring more security issues. We summarize them in Table 4.

Table 4: Security vulnerabilities in *fs* block management.

FS	Extent	Extended Attribute	Delay Allocation	Flex/Meta Block Group	Inline data
Ext4	9	4	1	5	2
XFS	2	5	-	-	-
F2FS	-	2	-	-	-
JFS	-	1	-	-	-
Total	11	12	1	5	2

4.5 Crash-safety Model

Crash consistency models like journaling [35], logging [31], shadow paging [12] are widely used in file systems. Take JFS as an example, it employs a synchronous writing strategy to log the storage operations and `inode`. Attackers could exploit the vulnerability in JFS to obtain the kernel information (`CVE-2004-0181`). Specifically, as JFS logs all relevant in-memory data structures, some of them may not be properly initialized. Attackers could obtain the sensitive kernel information by reading the raw device directly.

In addition, we find that vulnerability would occur when the crash-safety model does not cooperate well with other OS components like I/O scheduler (e.g., `CVE-2017-7495`). For example, the I/O scheduler may reorder inode block and data block. As a result, the inode block may come to the disk before the data block, which violates the semantic *ordered* journal model of JBD2. In this case, malicious users may read stale data from disk, which include other users' file content.

4.6 Semantic Mismatch with VFS

Virtual file system (VFS) provides the general data structures and interfaces for underlying *fs* implementations. Each specific *fs* implementation must obey the specification provided by the VFS. However, we find that there are many semantic mismatches between VFS and *fs* implementations (e.g., `CVE-2016-6198`, `CVE-2008-3275`). For example, if a file is renamed to a hardlink of itself, the underlying implementation of VFS – OverlayFS unifies multilple file systems and provides a hierarchical namespaces [28]. Different OverlayFS layer has different inode for the same namespace. Therefore, suppose a file is renamed to a hardlink of itself, VFS must retrieve the real inode via *d_select_inode* during *vfs_rename* procedure. Otherwise, VFS may proceed with incorrect inode, resulting in data loss. It is desirable to develop techniques to enforce the consistent assumption and implementation across the storage stack from the VFS to underlying real *fs*.

Summary: *We identify that fs vulnerabilities center around metadata management and data access permission checking. Particularly, metadata management is the most vulnerable component in file systems, which occupies 74% of the total studied vulnerabilities.*

5 EXPLOITING FS FEATURES

As we expected, we find that *fs* features could be exploited by adversaries. In this section, we use two case studies to demonstrate how adversaries could exploit the *fs* features to facilitate their attacks.

5.1 Malformed Disks

Due to the complicated disk layout in file systems, adversaries could leverage a crafted disk image to conduct evil maid attack. Most OS kernels automatically detect and mount the untrusted devices with supported file systems [40]. During the mounting procedure, only data fields in the superblock are sanity checked, other critical system structure (e.g., block metadata) are not verified. Consequently, the insane on-disk structures pose severe threats to the computer system. According to our study, substantial vulnerabilities (29 cases) are caused by mounting and manipulating malformed disk images. Recently, researchers propose solutions like Recon [9] to perform *fs* consistency check to avoid metadata corruption. However, they assume that the disk is well-formed and thus fail to detect malformed disk images.

5.2 Data Caching

Modern file systems use kernel page cache to improve application performance. They cache data blocks in the page cache for accelerating file accesses. We observe that page cache could be exploited by attackers to corrupt the disk data. For example, as reported in `CVE-2015-8839`, Ext4 provides *hole punching* to remove a portion of a file without changing its size. To fulfill this function, the corresponding data blocks resided in the page cache will be freed. However, Ext4 does not synchronize the page fault and hole punching procedure, leading to a race condition. In this case, page fault handler would map the pages containing the freed blocks into another process's address space. Thus, disk data corruption would happen if these pages are written to disk. To mitigate this issue, file systems should carefully handle the consistency between memory states and disk states.

Summary: *Adversaries do exploit fs features to facilitate their attacks, and most of these vulnerabilities are not easy to be detected. To defend against such attacks, file systems should integrate security as an intrinsic property to enforce essential checks in critical system components.*

6 RELATED WORK

Bug and vulnerability study. Previous research has conducted intensive studies on file systems bugs [21, 27], concurrency bugs [8, 11, 19, 23], Linux kernel vulnerabilities [4], virtual memory manager [14], and Linux malware [7]. Our work uses a similar study methodology. However, it focuses on the security vulnerability in Linux file systems. To the best of our knowledge, this is the first work conducting such a detailed study on the security vulnerability of file systems.

Bug and vulnerability detection. Fighting against *fs* bugs has attracted much attention recently. We classify existing research works into two categories: *debugging* and *verification*. Debugging tools like FiSC [43] and eXplode [41] can detect *fs* bugs based on the identified bug patterns. The patterns and insights derived from our study could inspire researchers to build efficient tools to detect *fs* vulnerabilities. Formal verification is another approach [1–3, 5, 10, 33]. However, it is still challenging to verify the correctness of an entire file system, due to their huge codebase [10, 33, 39]. Our study pinpoints the critical *fs* functions and components that suffer from the most vulnerabilities, which could help researchers narrow their codebase for verification.

Secure file systems. Beyond bug study and detection, researchers also aim to build secure and reliable file systems [22, 26]. Lu et al. [22] proposed physical disentanglement to minimize storage faults propagation. Min et al. [26] leveraged transaction flash storage to build a crash-consistent file system. Our vulnerability study would motivate researcher to enhance the security aspect of file systems and provide them insights to build secure file systems.

7 CONCLUSION AND FUTURE WORK

In this paper, we conduct an empirical study on the security vulnerabilities in popular file systems. Our findings reveal their existing and potential design and implementation flaws. We expect our study would motivate and inspire system researchers and developers to build more secure file systems and vulnerability-detection tools.

As the future work, we propose to extend such a vulnerability study to more popular file systems running on different computing platforms, such as distributed file systems and mobile file systems. Inspired by the insights provided in this paper, we plan to develop generic vulnerability-detection and verification tools that can address a category of security issues in file systems. And meanwhile, we will rethink the design and implementation of file systems to integrate the security as their inherent property.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CNS-1850317 and NetApp Faculty Fellowship Award.

REFERENCES

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby C. Murray, Gerwin Klein, and Gernot Heiser. 2016. CoGENT: Verifying High-Assurance File System Implementations. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016.* 175–188.

[2] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016.* 83–98.

[3] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* 270–286.

[4] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux kernel vulnerabilities: state-of-the-art defenses and open problems. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011.* 5.

[5] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* 18–37.

[6] Ann Chervenak, Vivekenand Vellanki, and Zachary Kurmas. 1998. Protecting file systems: A survey of backup techniques. In *Proceedings of Joint NASA and IEEE Mass Storage Conference.*

[7] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. 2018. Understanding Linux Malware. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA.* 161–175.

[8] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. 2010. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010.* 221–230.

[9] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: verifying file system consistency at runtime. In *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012.* 7.

[10] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 653–669.

[11] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, USA, November 03 - 05, 2014.* 7:1–7:14.

[12] Dave Hitz, James Lau, and Michael A. Malcolm. 1994. File System Design for an NFS File Server Appliance. In *USENIX Winter 1994 Technical Conference, San Francisco, California, USA, January 17-21, 1994, Conference Proceedings.* 235–246.

[13] Jian Huang, Michael R. Allen-Bond, and Xuechen Zhang. 2017. Pallas: Semantic-Aware Checking for Finding Deep Bugs in Fast Path. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17), Xi'an, China, April 8-12, 2017.*

[14] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. 2016. An Evolutionary Study of Linux Memory Management for Fun and Profit. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.* 465–478.

[15] Jian Huang, Jun Xu, Xinyu Xing, Peng Liu, and Moinuddin K. Qureshi. 2017. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 2231–2244.

[16] Jian Huang, Xuechen Zhang, and Karsten Schwan. 2015. Understanding Issue Correlations: A Case Study of the Hadoop system. In *Proceedings of ACM Symposium on Cloud Computing (SoCC'15), Kohala Coast, Hawaii, 2015.*

[17] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. 2003. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA.*

[18] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015.* 273–286.

[19] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. 2016. TaxDC: A Taxonomy of Non-Deterministic Concurrency Bugs in Datacenter Distributed Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016.* 517–530.

[20] Yan Li, Nakul Sanjay Dhotre, Yasuhiro Ohara, Thomas M. Kroeger, Ethan L. Miller, and Darrell D. E. Long. 2013. Horus: fine-grained encryption-based security for large-scale storage. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013.* 147–160.

[21] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A study of Linux file system evolution. In *Proceedings of the 11th USENIX conference on File and Storage Technologies, FAST 2013, San Jose, CA, USA, February 12-15, 2013.*

[22] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical Disentanglement in a Container-Based File System. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.* 81–96.

[23] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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, Seattle, WA, USA, March 1-5, 2008.* 329–339.

[24] Ethan Miller, Darrell Long, William Freeman, and Benjamin Reed. 2001. Strong Security for Distributed File Systems. In *2001 IEEE International Performance, Computing, and Communications Conference, IPCCC'01, Phoenix, AZ, USA, April 4-6, 2001.*

[25] Ethan L. Miller, Darrell D. E. Long, William E. Freeman, and Benjamin Reed. 2002. Strong Security for Network-Attached Storage. In *Proceedings of the FAST '02 Conference on File and Storage Technologies, January 28-30, 2002, Monterey, California, USA.* 1–13.

[26] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. 2015. Lightweight Application-Level Crash Consistency

on Transactional Flash Storage. In *2015 USENIX Annual Technical Conference, USENIX ATC '15, July 8-10, Santa Clara, CA, USA*. 221–234.

[27] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking semantic correctness: the case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. 361–377.

[28] Overlayfs. 2019. https://wiki.archlinux.org/index.php/Overlay_filesystem.

[29] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. 433–448.

[30] CVE Researcher Reservation Guidelines. 2019. https://cve.mitre.org/cve/researcher_reservation_guidelines.

[31] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*. 1–15.

[32] Andy Sayler and Dirk Grunwald. 2014. Custos: Increasing Security with Secret Storage as a Service. In *2014 Conference on Timely Results in Operating Systems, TRIOS '14, Broomfield, CO, USA, October 5, 2014.*

[33] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. 1–16.

[34] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. 2000. Self-Securing Storage: Protecting Data in Compromised Systems. In *4th Symposium on Operating System Design and Implementation (OSDI 2000), San Diego, California, USA, October 23-25, 2000*. 165–180.

[35] Stephen Tweedie. 1998. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo, 1998*.

[36] WannaCry Ransomware Attack. 2017. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack.

[37] Jake Wires and Michael J. Feeley. 2007. Secure file system versioning at the block level. In *Proceedings of the 2007 EuroSys Conference, Lisbon, Portugal, March 21-23, 2007*. 203–215.

[38] XFS. 2017. https://en.wikipedia.org/wiki/XFS.

[39] Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 59–79.

[40] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing File Systems via Two-Dimensional Input Space Exploration. In *2019 IEEE Symposium on Security and Privacy, SP 2019, Proceedings, 20-22 May 2019, San Francisco, California, USA*. 161–175.

[41] Junfeng Yang, Can Sar, and Dawson R. Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*. 131–146.

[42] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson R. Engler. 2006. Automatically Generating Malicious Disks using Symbolic Execution. In *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*. 243–257.

[43] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. 2004. Using Model Checking to Find Serious File System Errors. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004*.