The Security War in File Systems: An Empirical Study from A Vulnerability-centric Perspective

JINGHAN SUN and SHAOBO LI, University of Illinois at Urbana-Champaign, USA
JUN XU, University of Utah, USA
JIAN HUANG, University of Illinois at Urbana-Champaign, USA

This article presents a systematic study on the security of modern file systems, following a vulnerability-centric perspective. Specifically, we collected 377 file system vulnerabilities committed to the CVE database in the past 20 years. We characterize them from four dimensions: why the vulnerabilities appear, how the vulnerabilities can be exploited, what consequences can arise, and how the vulnerabilities are fixed. This way, we build a deep understanding of the attack surfaces faced by file systems, the threats imposed by the attack surfaces, and the good and bad practices in mitigating the attacks in file systems. We envision that our study will bring insights towards the future development of file systems, the enhancement of file system security, and the relevant vulnerability-mitigating solutions.

CCS Concepts: • **Security and privacy** → **File system security**; • **Information systems** → **Storage management;**

Additional Key Words and Phrases: Storage systems, file system vulnerabilities

ACM Reference format:
Jinghan Sun, Shaobo Li, Jun Xu, and Jian Huang. 2023. The Security War in File Systems: An Empirical Study from A Vulnerability-centric Perspective. *ACM Trans. Storage* 19, 4, Article 34 (September 2023), 26 pages.
https://doi.org/10.1145/3606020

1 INTRODUCTION

After decades of development since the 1960s, file systems have become a core component in nearly all computer systems. Working as the direct interfaces to user data and typically running at a high privilege, file systems are highly security-sensitive. This has incentivized the creation of various techniques to harden the security of file systems, ranging from formal verification [21, 23, 75, 91] to access control [31, 77] and sanity checking [35, 42, 54].

Despite the above efforts, the security of file systems is still frequently compromised [9, 76], causing damages such as data breaches and hijacked execution. In addition, attackers often exploit vulnerabilities in file systems to launch ransomware attacks, affecting hundreds of victims and costing tens of millions of dollars [81, 93]. To gain insights towards changing this situation, it is

This work was partially supported by a NetApp Faculty Fellowship Award and NSF CAREER Award 2144796.

Authors' addresses: J. Sun, S. Li, and J. Huang, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA, 61820; emails: {js39, shaobol2, jianh}@illinois.edu; J. Xu, University of Utah, Salt lake City, Utah, USA, 84102; email: junxzm@cs.utah.edu.

1553-3077/2023/09-ART34 $15.00
https://doi.org/10.1145/3606020

beneficial to build a systematic understanding of the security of file systems from angles such as the attack surfaces [60, 85] in file systems and the threats imposed by the attack surfaces. Past studies [49, 64] on file systems mostly focus on functionality-related perspectives, failing to provide the desired security-centric understanding. Prior studies [16, 49] made an initial effort to investigate the filesystem bugs and security in Linux. However, they targeted solely generic file systems, with a focus on the causes and consequences, which lacks an analysis of the whole lifecycle of the vulnerability exploitation and fixes. We believe that a thorough understanding of the attack exploitation procedure in file systems could help developers to advance the design principles and implementation strategies of file systems.

In this article, we present an in-depth study on the security of various file systems, aiming to fundamentally understand the entire vulnerability exploitation procedure and possible fix strategies. Specifically, our study centers around addressing the following questions:

- *Q1: What are the major attack surfaces in file systems?*
- *Q2: What are the threats posed by the attack surfaces?*
- *Q3: What are the common approaches to exploiting these attack surfaces by attackers?*
- *Q4: How are the attack surfaces fixed?*

Methodology: To answer the above questions, our study takes a vulnerability-centric approach with a focus on the reported file system vulnerabilities. We examined the full exploitation paths of vulnerabilities, including their root causes, exploitation techniques, and the methods used to fix them. In total, we collected 377 file system vulnerabilities committed to the CVE database [33] in the past 20 years. They reported explicit security implications and covered nearly all the mainstream file systems (*generic file systems* such as Ext4 [52], XFS [82], and F2FS [44], *mobile file systems* such as HFS [38] and APFS [12], and *networked file systems* such as NFS [58], GlusterFS [29], and Ceph [19]).

We manually analyzed the 377 vulnerabilities and sought answers to questions *Q1* to *Q4*. First, we investigated the root causes of the vulnerabilities by analyzing the vulnerable code and the corresponding patch(es). We then categorized the vulnerabilities based on their root causes. This enabled us to gain an understanding of the attack surfaces opened by the filesystem vulnerabilities and the characteristics of the attack surfaces (*answering Q1*). Second, we manually and statically reasoned the possible execution paths to exploit the vulnerabilities. In this process, we summarized the potential threats that adversaries can bring by following those exploiting paths (*answering Q2 and Q3*). Finally, we surveyed the patches to the vulnerabilities, unveiling the good and bad practices in mitigating the attack surfaces (*answering Q4*). Throughout the study, we derived a set of findings. We summarize the major ones that help answer *Q1–Q4* as follows.

Attack surfaces: Vulnerabilities in file systems are diverse. Although 27% are general types of vulnerabilities that may appear in other software systems (e.g., memory errors and race conditions), a unique cause of these general vulnerabilities is the addition of new features, such as crash-consistency models into file systems. The remaining vulnerabilities are unique to file systems but attributed to different reasons:

- 29% are rooted in sanity checks that are used to validate the states of various file system properties (e.g., namespace, inode attributes, and file category). The sanity checks are highly semantic related, whose implementation is complicated and often leads to vulnerabilities.
- 15% are due to the lack of permission checks or incorrect access control. Similar to sanity checks, permission checks and access control are semantic dependent and highly complex. Their buggy implementations largely turn into permission- and access-related vulnerabilities.

- 18% are introduced when the functionality is extended to support the use environments of specific file systems. For instance, many vulnerabilities appear in encryption mechanisms adopted by mobile file systems to defend against physical attacks.

Threats and their exploitation procedure: File systems vulnerabilities are dangerous. As we will illustrate later in Figure 6, file system vulnerabilities pose a similar security threat level to that of vulnerabilities in other OS components. The specific consequences after exploiting file system vulnerabilities vary across vulnerability types.

- Exploiting general types of vulnerabilities (e.g., memory errors) in file systems can largely cause their common sequences, such as denial-of-service (DoS) and hijacked execution. An interesting observation is that general vulnerabilities in file systems can more often lead to data leakage because file systems directly deal with user data.
- Exploiting erroneous sanity checks can also lead to DoS but more often data leakage, which allows attackers to initiate further attacks such as privilege escalation and permission bypass.
- Exploiting lack of permission checks or incorrect access control produces more predictable consequences, often including permission bypass and privilege escalation as well as subsequent damages such as data leakage and data corruption.
- Exploiting the remaining vulnerabilities typically causes consequences pertaining to the use contexts of the file systems. For instance, exploiting vulnerabilities in the encryption modules of mobile file systems often enables physical attacks, whereas exploiting vulnerabilities in the networking management of network file systems usually leads to code injection attacks.

Fixes with patches: File system patches are diverse in both complexity and functionality—56% of the patches only change less than 10 lines of code, using simple operations such as adding a boundary check and fixing a syntax error (see the details in Figure 8). In contrast, around 20% of the patches bring modifications to over 50 lines of code, involving complex behaviors ranging from full validation of data structures to fixes in multiple components. The majority of existing and current patches work in an ad-hoc style. For instance, the patches to many race conditions simply make the responsible operations atomic. These ad-hoc patches can heal the target vulnerabilities, but they fail to prevent similar vulnerabilities from emerging or fundamentally mitigate them. Nevertheless, our study identifies positive signs. Strategies to enforce preventive and secure-by-design patching are arising. For instance, many patches of race conditions switch from lock-based implementations to lock-free implementations, fundamentally eliminating issues such as deadlocks. File systems are also often correlated, as are their patches. In particular, a patch to the VFS subsystem often requires updating the underlying file systems. Otherwise, the patch may not come into effect. However, today's practice often overlooks the correlated file systems when patching the target ones, thus leading to insecure patches.

Contributions: Our main contributions are as follows.

- We perform the first in-depth study on the security of file systems from a vulnerability-centric perspective. The study unveils the attack surfaces of file systems, followed by demystifying the root causes and threats of the attack surfaces.
- We present 21 previously unknown findings throughout the study. Besides facilitating a deeper understanding of the attack surfaces, the findings reveal how the development of file systems influences or even impedes security.
- We distill 7 insights from the results and findings of the study. The insights point out the deficiencies in today's practice of addressing security threats faced by file systems and also bring up a set of guidance towards improvements.

Table 1. File Systems Covered in Our Study[1]

Type	**File Systems**	**# of CVE**
Generic	Ext4, VFS, XFS, F2FS, BtrFS, profs, Ext2, Ext3, tmpfs, ReiserFS, JFS	178
Mobile	HFS/HFS+, APFS, F2FS, eCryptFS	57
Networked	NFS, OpenAFS, GlusterFS, CephFS, HDFS	142
Total	–	377

2 STUDY METHODOLOGY

Our study aims to understand the attack surfaces in file systems, from the perspectives summarized as *Q1–Q3*. To support the study, we consider public vulnerabilities as the data source. In comparison with other applicable data sources such as attacks that happened in the past, vulnerabilities are more accessible and more abundant.

2.1 Data Collection

We collected file system vulnerabilities from the CVE database [33]. We considered the CVE database for three reasons. First, the CVE reports are reviewed and confirmed by trustworthy parties (major IT vendors, security companies, and research organizations [7]) and, thus, offer high reliability. Second, the CVE reports largely provide important details, such as root cause information and patch information. Such details are helpful for us to answer *Q1–Q3*. Third, the CVE reports are large in number and diverse in terms of sources, which should properly reflect the overall situation.

Specifically, we examined the CVE reports committed in the past 20 years (January 1999 to December 2020) and identified those associated with mainstream file systems. They cover most of the widely deployed open-sourced file systems, as illustrated in Table 1. Older reports were omitted since they may not represent what is happening at present. We further refined the reports to only keep those carrying information about or giving references to the following three **criteria**: (**i**) location of the vulnerable code, (**ii**) analysis of the root cause (comprehensive or brief), (**iii**) possible exploitation of the vulnerability and subsequent consequences (which will be verified following our approach in Section 2.2). We present below an example CVE report (`CVE-2018-1094`) that includes the three aforementioned criteria:

> *The ext4_fill_super function in fs/ext4/super.c* **(criterion i, location of code)** *in the Linux kernel through 4.15.15 does not always initialize the crc32c checksum driver* **(criterion ii, root cause)**, *which allows attackers to cause a denial of service* **(criterion iii, consequence)** *via a crafted ext4 image.*

In total, we filtered out 35 CVEs, and kept 377 CVEs that cover the three criteria. We present their distribution across time in Figure 1. We also collected the patches to these CVEs to further understand the strategies used to fix the vulnerabilities. Among all CVEs, 62% of them have open-sourced their patches, which facilitates our analysis.

2.2 Vulnerability Analysis

We ran three analyses on each of the 377 vulnerabilities.

[1]VFS (Virtual File System) [4] is a software layer that lies between the operating system kernel and the actual file system implementations, providing a unified interface to access different file systems. A VFS vulnerability is not exploited alone but rather together with its underlying generic file systems. NFS (Network File System [58]) is a network protocol that allows computers to access files over a network. In our study, we cover both of its NFSv3 and NFSv4 implementations.

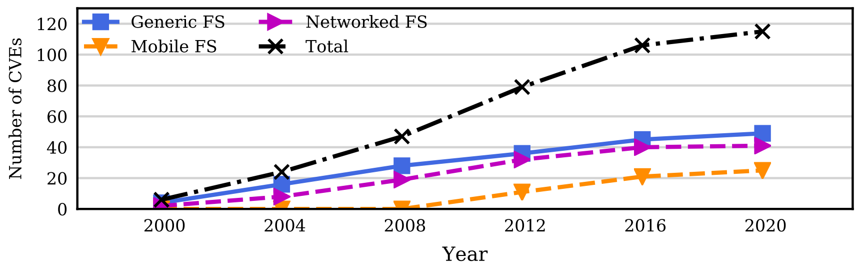


Fig. 1. Distribution of vulnerabilities over the past 20 years.

```
1 int xfs_ioc_fsgetxattr(xfs_inode_t *ip,int attr,...){
2   struct fsxattr fa;
3 + memset(&fa, 0, sizeof(struct fsxattr));
4   // data structure ``fa'' is uninitialized before patching
5   xfs_ilock(ip, XFS_ILOCK_SHARED);
6   fa.fsx_xflags = xfs_ip2xflags(ip);
7   if (copy_to_user(arg, &fa, sizeof(fa)))
8     return -EFAULT;
9 }
```

Fig. 2. An uninitialized memory vulnerability in XFS and its patch (CVE-2010-3078).

Root cause analysis. The first analysis focuses on understanding the root cause of a CVE report. In this analysis, we first extract descriptions of the root causes from the report and then verify the description through interpretation of the vulnerable code and its patch.

Exploitation analysis. Following the root cause analysis, we aim to understand the possible exploitation of a vulnerability and the subsequent consequences. One common approach is to reproduce the existing proof-of-concept (PoC) exploits against the vulnerability [56]. These PoC exploits trigger simulated attacks with test programs and reproduce the consequences to expose the system weakness. However, PoC exploits are not available for most of the CVE reports due to security concerns.

In our study, we use a more generic and lightweight approach. We first parse the description in a CVE report to retrieve the reported exploitation and its consequences. We then reason the vulnerable code to identify execution paths that can verify the description. Specifically, an execution path successfully verifying the exploitation must meet the following conditions:

- The execution path covers a feasible route from a user-space interface to the vulnerable code.
- The execution path covers a feasible route to triggering the vulnerability.
- The execution path covers a feasible route from triggering the vulnerability to causing the described consequence(s).

If no such execution paths can be identified for a CVE report, we skip the report. The example in Figure 2 illustrates our approach. In the example, a local data structure *fa* (declared at line 2) is not fully initialized before it is copied to user space at line 7. The CVE report says *"It allows local users to obtain potentially sensitive information from kernel stack memory via an ioctl call"*. We find that any execution path first opening a legitimate file in an XFS file system and then calling ioctl (with XFS_IOC_FSGETXATTR) on the opened file will trigger the vulnerability, and leak data from the kernel stack. This way, the exploitation and the consequence are considered verified.

Patch analysis. The last analysis examines the patches to file system vulnerabilities. One goal is to understand the strategies used by the existing patches (e.g., ad-hoc or systematic). This way, we expect to unveil whether today's practice of file system patching can help fundamentally resolve the issues of vulnerabilities. Another goal of the analysis is to summarize the aspects overlooked by file system developers when patching vulnerabilities. We anticipate bringing related evidence to incentivize broader attention to those aspects.

2.3 Discussion of Study Validity

While we carefully designed our methodology, our study may have several limitations. First, similar to other sampling-based studies, our study can miss certain types of vulnerabilities. Thus, our findings might be biased towards the collected samples. To mitigate the issue, we include as many types of popular file systems as possible in our CVE dataset, from generic file systems ext4 [52] to networked file systems GlusterFS [29] and HDFS [37]. Our study also does not aim to discover uncaught vulnerabilities in file systems. Second, we verify the exploitation of a vulnerability and the consequences through static reasoning. Unlike PoC exploits, static reasoning does not guarantee the fidelity of the results. As such, we may present underestimated/exaggerated/erroneous threats of the vulnerabilities. To mitigate this issue, we document the execution paths identified to exploit the vulnerability and then have another person review and vet the execution paths. Any execution paths that raise discrepancy are discarded. Third, the analysis of the patches can often require deep domain knowledge. Lack of such knowledge will lead to misunderstanding about the patches. To mitigate the problem, we only allow authors whose research focus is file/storage systems to look at the patch and again, we have multiple authors vet the results.

Similar to prior characteristic studies that may suffer from limitations of sampling, we make our best effort to collect the available vulnerabilities. Given that we focus on popular file systems developed over the past two decades, we believe that the limitations do not invalidate our study results. We also encourage readers to focus on the attack procedure behind each case rather than the precise numbers, since a single vulnerability can cause massive severe damages.

3 TAXONOMY OF FILE SYSTEM VULNERABILITIES

We present the breakdown of file system vulnerabilities in Table 2, and their distribution over time in Figure 1. Popular file systems tend to have more CVEs. For example, the popular Linux file system ext4 [52] has the largest number of vulnerabilities reported. To facilitate our study, we first build a deeper understanding of the categories of vulnerabilities. We categorize the vulnerabilities into two types:

- *Generic vulnerabilities* – vulnerabilities that can appear in any kind of software systems. The major types include memory errors and race conditions.
- *Semantic vulnerabilities* – vulnerabilities that are unique to file system design and implementation. These vulnerabilities are mostly caused by errors related to file system semantics, such as violations to the file system permission model or missing sanity checks to the metadata structures.

The categorization brings a set of statistical findings and interesting observations. We summarize our findings as follows:

F-1: Generic vulnerabilities are common in file systems (>27%) but present positive signs. And there has been a misconception that generic vulnerabilities are decreasing as file systems become mature. Our study reveals that there would be a significant amount of deep memory and concurrency vulnerabilities that can be detected with advanced bug-finding tools, as shown in Table 2. Generic vulnerabilities account for a significant portion of the vulnerabilities

Table 2. Summary of File System Vulnerabilities

File Systems				Vulnerabilities					
Type	**Name**	**Release Time**	**#CVEs**	M	C	S	P	N	O
Generic	JFS [2]	1990	4	50%	0%	50%	0%	0%	0%
	Ext2 [18]	1993	7	14%	14%	43%	14%	0%	14%
	XFS [82]	1993	23	30%	4%	35%	4%	0%	26%
	VFS [4]	1995	28	18%	11%	40%	18%	0%	14%
	procfs [55]	1999	18	22%	6%	11%	22%	0%	39%
	ReiserFS [5]	2000	5	40%	0%	40%	20%	0%	0%
	Ext3 [80]	2001	11	9%	0%	36%	18%	0%	36%
	tmpfs [78]	2001	6	17%	0%	17%	33%	0%	33%
	Ext4 [52]	2008	45	13%	4%	49%	4%	0%	24%
	Btrfs [68]	2009	13	15%	8%	38%	31%	0%	8%
	F2FS [44]	2012	18	11%	11%	56%	6%	0%	17%
	Total		178	19%	6%	39%	13%	0%	22%
Mobile	HFS/HFS+ [38]	1985	8	25%	0%	75%	0%	0%	0%
	eCryptFS [36]	2006	16	44%	0%	25%	19%	0%	13%
	F2FS [44]	2012	11	36%	0%	45%	9%	0%	9%
	APFS [12]	2016	4	25%	25%	25%	25%	0%	0%
	Others	–	18	22%	17%	33%	6%	0%	22%
	Total		57	32%	7%	39%	11%	0%	12%
Networked	NFS [63]	1984	72	17%	7%	13%	20%	24%	19%
	GlusterFS [29]	2005	18	16%	0%	11%	11%	39%	22%
	OpenAFS [3]	2006	27	44%	4%	11%	11%	22%	7%
	HDFS [37]	2006	6	0%	0%	33%	67%	0%	0%
	CephFS [84]	2012	19	16%	0%	16%	21%	26%	21%
	Total		142	21%	4%	13%	19%	25%	16%

M: Memory Errors; *C:* Concurrency Issues; *S:* Sanity Check Errors; *P:* Permission Errors; *N:* Network Errors; *O:* Others (e.g., hash collisions such as `CVE-2014-7283`).

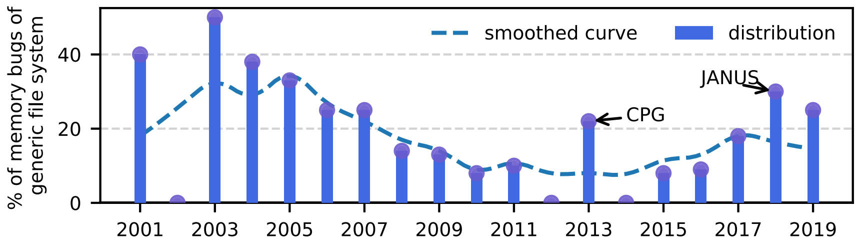


Fig. 3. Distribution of memory errors reported from file systems over time. The deployment of JANUS [88] and code property graph (CPG) [89] led to spikes of reported memory bugs in 2013 and 2018.

in file systems. In particular, memory errors and concurrency issues led to 21.7% and 5.5% of all the vulnerabilities, respectively. The numbers are not surprising since file systems are largely developed in memory unsafe languages (e.g., C/C++) and widely support concurrency.

There is no doubt that the above numbers give the important message that generic vulnerabilities still constitute a major attack surface in file systems. We cannot overlook generic vulnerabilities, since we do observe a gradual rise in memory errors after 2012 (see Figure 3). Various bug-finding tools (e.g., JANUS [88] in 2018 and Code Property Graph [89] in 2013) also emerged in recent years. Real-world deployment of these tools since 2012 has been disclosing more generic vulnerabilities in file systems. Consider Figure 3 as an example. Code Property Graph [89] and

JANUS [88], after deployment, discover a surge of memory errors in generic file systems. JANUS applies fuzzing techniques to file system vulnerability detection by mutating file system images and manipulating file system operations as the input to the fuzzer. The large search space of fuzzing helps detect deep memory errors that cannot be manually tested via hand-crafted images (e.g., `CVE-2018-10880`). This could encourage researchers to apply other advanced bug detection tools to file system vulnerabilities.

`F-2`: Semantic vulnerabilities are the dominant category of vulnerabilities in file systems (43%). File systems involve highly complex semantics for reliable and efficient file management. Implementation of the semantics is challenging and often error-prone, bringing the dominant category of vulnerabilities. There are two major sources leading to semantic vulnerabilities, sanity checks and file permissions. We separately discuss them in the following.

Sanity checks. File systems maintain a mass of semantic-related states, such as file system namespace, inode attributes, cache consistency, and so on. Missing sanity checks on the states is a major way of introducing vulnerabilities. For instance, the implementations of the journaling in Ext2 and Ext3 never validate the journal superblock before reading it, which can cause an assertion or even kernel corruption (`CVE-2011-4132`). Erroneous sanity checks on the states is another way of introducing vulnerabilities. For example, the f2fs utilities before version 1.12.0 has a logical flaw when validating the sanity of the superblock (`CVE-2020-6070`). The logical flaw can lead to bypass of validation and even user-controlled execution. In total, missing sanity checks and erroneous sanity checks caused 29% of the total vulnerabilities in file systems.

File permissions. File permissions are indispensable to the protection of user data. To enforce file permissions, file systems rely on standard permission models (e.g., `u/g/o:r/w/x`[2]) and access control list (ACL) (e.g., POSIX ACLs) to manage accesses. The actual implementations of permission models and ACL, however, often miss or add insufficient permission checks, leading to 15% of the vulnerabilities in file systems. For example, ReiserFS misses permission checks on the accesses to the directory storing file attributes (i.e., `xattrs`) (`CVE-2010-1146`). This enables any user to access or modify the attributes of files that are not allowed.

In contrast to generic vulnerabilities, semantic vulnerabilities are more concerning today. On the one hand, most of the semantic vulnerabilities seem to be discovered in an ad-hoc manner. We did not identify many tools that can systematically detect semantic vulnerabilities. While some generic tools, such as the fuzzing-based JANUS, do help find semantic vulnerabilities, they are still better at detecting generic vulnerabilities as they often lack domain-specific guidance to capture semantic vulnerabilities. On the other hand, as shown in Figure 4, the number of reported semantic vulnerabilities is clearly increasing over time. This might be attributed to the fact that more file systems are included in our study as time moves forward or that the number of hidden semantic vulnerabilities is significantly large. Either way, the trend implies an increasing threat posed by semantic vulnerabilities in file systems.

Summary: Both generic vulnerabilities and semantic vulnerabilities open large attack surfaces in file systems. However, generic vulnerabilities preset a seemingly improving trend while semantic vulnerabilities show the opposite.

Insight-1: To heal the attack surfaces of file systems, more efforts should be prioritized to mitigate semantic vulnerabilities. In particular, it is wise to explore tools that can systematically unveil semantic vulnerabilities.

[2]In the notation, "u", "g", and "o" represent three user classes: the owner user, the users in the file's group, and other users, respectively. Permission-symbols "r", "w", and "x" represent permission settings of three operations: "read", "write", and "execute", respectively.

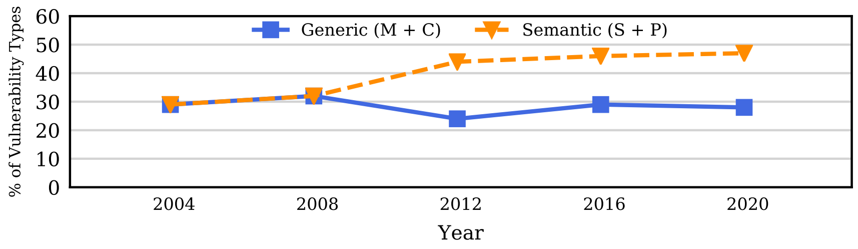


Fig. 4. Distribution of vulnerabilities across time. Generic vulnerabilities are stabilizing and semantic vulnerabilities present an increasing trend.

Table 3. Distribution of Vulnerabilities Caused by Different Features in Block Management

File Systems	#CVEs	Extent	Extended Attribute	Delayed Allocation	Flex/Meta Block Group	Inline Data
Ext4 [52]	45	29%	13%	4%	16%	6%
XFS [82]	23	6%	16%	-	-	-
F2FS [44]	18	-	6%	-	-	-
JFS [2]	4	-	25%	-	-	-

Only representative file systems are selected. The numbers are per file system.

Our categorization brings knowledge about the major vulnerabilities to be addressed. However, it gives few insights into how. To this end, we further investigate which file system functionality and what file operations introduce the vulnerabilities. In a nutshell, every major functionality of file systems brings in vulnerabilities, but inode management, block management, and page cache system bring in more. And a key observation is that extending the existing core functionality with new features contributed to a substantial number of vulnerabilities. We detail our findings below.

F-3: Inode management is a major source of introducing vulnerabilities (13% out of Linux FS). As per our study, inode management is also one of the most vulnerable parts of file systems. One reason is the complexity in the nature of inode management. Another reason, which is becoming more dominant, is the increasing need to support newly emerging features such as inline data blocks and extended attributes. A particular example is Ext4. Unlike Ext2 and Ext3, which uniformly keep extended attributes in a block, Ext4 supports three on-disk formats to store extended attributes, including internal inode, additional block, and dedicated inode. In regular cases where the extended attributes are small, Ext4 simply stores them in the internal inode. However, when the attributes grow bigger, Ext4 will swap them to an additional block. This type of format adjustment brings extra complexities and causes multiple vulnerabilities (e.g., `CVE-2018-11412`). Prior work proposed isolating metadata structures of unrelated files and directories to avoid failure propagation [50]. Inode replication was also proposed to ensure its fault tolerance [6]. We believe that more efforts should be made to decouple the *inode* management from other components in file systems.

F-4: Block management incurs a significant portion of vulnerabilities, and presents an increasing trend. These vulnerabilities could be mitigated by having a more expressive interface to offload block management to the storage devices. Block management accounts for 21% of Linux file system vulnerabilities. In Table 3, we present the ratio of vulnerabilities caused by different features in block management. This is not surprising since block management has complexity comparable to inode management. Moreover, more features are being integrated into block management for better efficiency. In particular, recent file systems, such as Ext4 and BtrFS, use extents instead of direct/indirect blocks to organize file blocks. The codebase of extent management (7K lines of code) is much larger than that of direct/indirect block organization (1.5K lines of code). The need for fast indexing, flexible block group [25], and delayed block allocation all contribute to the increasing complexity of the block management and, hence, result in more vulnerabilities. Instead of purely relying on the storage software for the block management, we believe that future file systems should explore an expressive interface to offload the block management tasks to storage devices. This could not only ease the file system development and verification procedure but also isolate its vulnerabilities from the OS kernel. As the hardware resources (i.e., storage processor and memory capacity) inside storage controllers become more powerful, the device is capable of handling more management tasks. The most recent work on device-level file systems, such as DevFS [41] and KEVIN [43], demonstrated the feasibility of offloading the file system functionalities to storage devices; however, they purely focused on performance improvement. It is highly desirable to explore the design trade-offs with respect to security enhancement.

`F-5`: Page cache causes 12% of the vulnerabilities and they could leak sensitive kernel information. Modern file systems widely use page cache to temporarily store data blocks that will be flushed to the persistent storage. The use of page cache avoids frequent interactions with the persistent storage and, thus, improves the performance of applications. However, the extra layer of cache creates new space for vulnerabilities. A major type of such vulnerabilities is caused by uninitialized cache. Specifically, developers often do not initialize many fields of metadata structures saved in page cache. Later, when the page cache is flushed, the uninitialized fields, whose memory may still carry sensitive kernel information, will be leaked to the persistent storage (e.g., `CVE-2005-0400 and CVE-2006-6054`). Any user who has access to the block can steal the sensitive information. As we will discuss in Section 4, uninitialized memory in cache systems poses higher threats than uninitialized memory in general because the former may always propagate to the disk at the point of flush. Therefore, the memory safety for page cache is especially critical in file systems.

`F-6`: Crash-consistency models for persistent storage often lead to uninitialized memory. Many file systems use crash consistency models, such as journaling [80], logging [69], and shadow paging [39]. For instance, JFS employs a synchronous writing strategy to log the storage operations and `inode`. This will ensure that the file system can always recover to a correct state at a crash. The support of crash-consistency introduces new data structures and requires sophisticated synchronizations across different components, which involves high complexity and often incurs vulnerabilities. For example, JFS logs all relevant in-memory data structures but many of them may not be properly initialized. The uninitialized data may later be written to persistent storage, resulting in leakage of kernel information (e.g., `CVE-2004-0181`).

`F-7`: The mismatch between VFS semantics and file system implementations introduces vulnerabilities. The correctness of VFS is crucial to the underlying file system. However, it attracts much less attention than the integrity checking of the actual file system implementations. Operating systems often incorporate a Virtual File System (VFS), which offers uniform interfaces for user applications and redirects operations from applications to the underlying file system. For things to work correctly, the underlying file system must follow the specifications of the VFS. This is, however, often violated and brings many vulnerabilities. VFS vulnerabilities (such as `CVE-2015-1420 and CVE-2015-2925`) could affect a wide range of file system implementations. Many fault-tolerant file system designs, such as EnvyFS [14], also rely upon the correct operations of the VFS layer. However, existing work mostly focused on the bug detection for each individual file system (e.g., ext4, btrfs) [42, 64, 88], and they rarely work on the correctness of the VFS layer [27]. Therefore, we believe that applying bug detection techniques such as model checking and fuzzing to VFS could also significantly improve file system security.

`F-8`: Functionalities for supporting use context can become the attack vectors of mobile and networked file systems. Unlike generic file systems, mobile file systems and networked file systems need special functionalities to support their use contexts. Implementation of these functionalities often involves vulnerabilities, which we detail below.

Mobile file systems typically run on portable devices such as smart phones, which can easily get lost and encounter physical attacks. To protect user data under physical attacks, many mobile file systems introduce encryption mechanisms. For instance, APFS leverages AES-XTS or AES-CBC to encrypt user files on demand [1]. F2FS, instead of directly providing file encryption, utilizes a cryptographic stacked file system (eCryptFS) to encrypt data. However, vulnerabilities can arise in the implementation, often breaking the security of the encryption mechanisms. APFS leaks encryption keys via the Disk Utility hints (`CVE-2017-7149`) and APFS allows reading of decrypted data via Direct Memory Access (`CVE-2017-13786`). The eCryptFS on Samsung KNOX 1.0 uses a weak key generation algorithm, which makes brute-force attacks feasible (`CVE-2016-1919`).

Networked file systems need communication modules based on protocols such as HTTP and RPC to support remote data transfer and access authentication. Many vulnerabilities appear in the communication modules due to mishandling of malformed or crafted network packets, such as zero payloads, crafted packet header, and uninitialized packets. For example, GlusterFS allows RPC requests to create symbolic links (e.g., `gfs3_symlink_req`) pointing to file paths in the GlusterFS volume. This enables adversaries to send crafted RPC to create arbitrary symbolic links in the storage server and then execute arbitrary code (`CVE-2018-10928`).

Summary: Core components in file systems, due to their high complexities, contribute a majority of the vulnerabilities. The situation is worsening because new features are being integrated into the core components and are building up the complexities.

Insight-2: Different file system components introduce different types of vulnerabilities, which are often related to their own semantics. This suggests that different tools should be tailored, in a semantic-aware manner, to detect vulnerabilities in different components. For instance, to detect uninitialized memory in cache systems, a tool needs to identify not only explicit access (e.g., by regular file system operations) but also implicit access (e.g., by synchronization of cache management).

Insight-3: At present, security seems insufficiently considered when new file system features are added. It is desirable that the trade-off between functionality/efficiency and security is taken into account before new features are incorporated, particularly in security-sensitive sectors. References about which new features bring what security issues can be found from **`F-3`** to **`F-8`**. To fundamentally improve the situation, proactive measures, such as regression test [8, 61, 92] on the new features, must be taken.

4 THREATS OF FILE SYSTEM VULNERABILITIES

Our taxonomy brings insights towards mitigating vulnerabilities in file systems. However, file system vendors or distributors may still hesitate to take action because the threats of the vulnerabilities are unclear to them. In this section, we summarize the threats posed by file system vulnerabilities. We anticipate bringing evidence to motivate mitigation and bring insights into prioritizing the mitigation against more dangerous vulnerabilities.

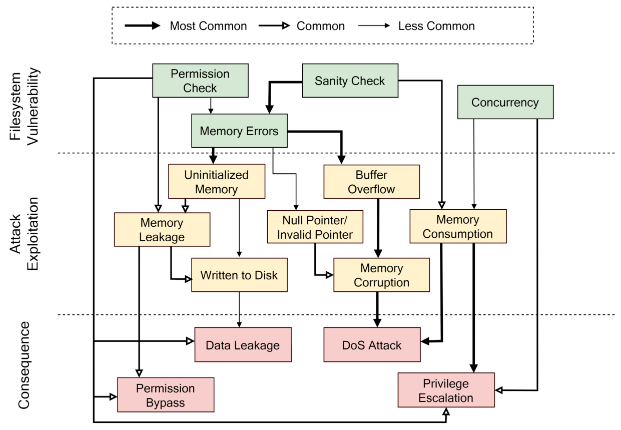
Overall, different vulnerabilities, when exploited, can lead to very diverse consequences. Figure 5 gives an overview of the trajectories from vulnerabilities in different file systems. The major attacks of generic file systems are caused by incomplete sanity checks, which exploit memory errors such as memory corruption and memory contention, which often result in Denial-of-Service (DoS). Mobile file systems have their unique attack paths, which mainly exploit the physical access vulnerabilities. They could bring severe consequences, such as data leakage. As networked file systems launch multiple file system instances, due to their incomplete packet filtering and validation, their network connection could be attacked with crafted packets via HTTP or RPC protocols. We also examine the distribution of vulnerabilities based on the threats they can pose in Table 4. For example, DoS is a major threat to both generic and networked file systems. A significant portion of vulnerabilities in mobile file systems can lead to privilege escalation. Table 5 demonstrates the correlation between the vulnerability types (see Section 3) and the threats. Following a similar organization of Section 3, we summarize our major findings below.

`F-9`: File systems vulnerabilities impose a comparable level of security threat to vulnerabilities in other OS components. Similar to our study on file system vulnerabilities, we gathered vulnerabilities in other operating system (OS) components that are committed to the CVE database in the past 20 years. As illustrated in Figure 6, file system vulnerabilities are as dangerous as vulnerabilities in other components, considering severity, impact, and exploitability as the metrics.

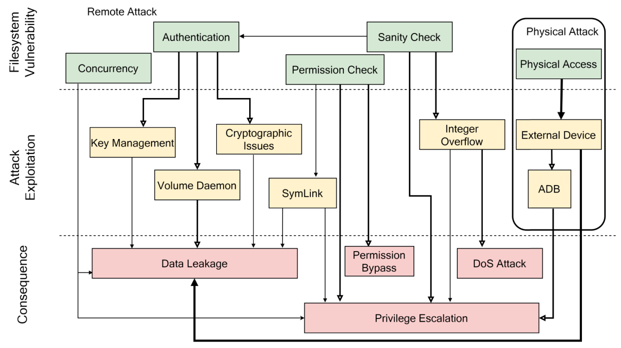
`F-10`: DoS is the dominant threat brought by vulnerabilities to file systems. About 61% of all the vulnerabilities can lead to DoS. As shown in Table 5, DoS is also the most common threat among memory errors, sanity check errors, network errors, and concurrency vulnerabilities. They often first incur one of the following behaviors and eventually cause a DoS.

- *Memory corruption*: Memory errors and race conditions in file systems, as they typically do, can often cause memory corruption, followed by a kernel panic, such as the case `CVE-2013-6382`. Many semantic vulnerabilities can also lead to memory corruption and subsequent kernel panics. The previously discussed `CVE-2015-8839` is such an example.
- *Memory consumption*: Another major cause of DoS in file systems is excessive memory consumption. Most of such cases are caused by memory leakage vulnerabilities. For instance, XFS before Linux 4.5.1 does not properly free the memory allocated for extended file attributes (`CVE-2016-9685`), which can be exploited to exhaust the memory and cause a DoS.
- *System hang*: CPU overconsumption, mainly brought by infinite loops and deadlocks, can turn into DoS. For instance, the integer overflow reported in `CVE-2017-18257` can be exploited to trigger an infinite loop in the block allocation of F2FS, causing a system hang.
- *Network congestion*: Vulnerabilities derived from protocol flaws, which account for about 16% of all the vulnerabilities in networked file systems, frequently trigger DoS. As shown in Table 5, 57% of the network errors can lead to DoS. Such vulnerabilities typically happen because of mishandling of malformed network packets, which often easily disrupt normal communication and result in DoS. For example, the remote procedure call (RPC) module in OpenAFS (before 1.6.23 or 1.8.x before 1.8.2) does not restrict the size of an input. Adversaries can send, or claim to send, large inputs and consume server resources waiting for those inputs, blocking service to other valid connections (e.g., `CVE-2018-16949`).

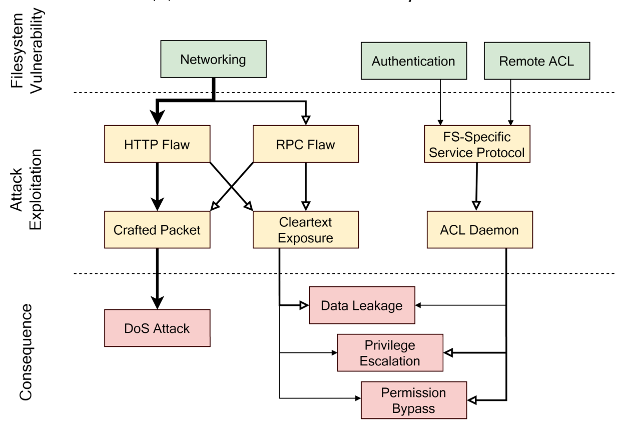
`F-11`: Data leakage is another big threat that vulnerabilities pose to file systems. Three major types of file system vulnerabilities often lead to data leakage. The first type is uninitialized



(a) Threats to generic file systems



(b) Threats to mobile file systems



(c) Threats to networked file systems

Fig. 5. Trajectory from vulnerabilities to threats in different types of file systems. For each type of file system, we present a graph of its vulnerability exploitation trajectories. Each trajectory represents a feasible route to triggering the vulnerability, which includes the root cause (e.g., sanity check), the exploitation methods, and the consequence (e.g., DoS attack). The bolded lines represent the most common trajectories.

Table 4. Distribution of File System Vulnerabilities Based on Their Threats

Type	Name	#CVEs	D	L	E	A	O
Generic	JFS [2]	4	50%	25%	0%	25%	0%
	Ext2 [18]	7	57%	14%	14%	14%	0%
	XFS [82]	23	70%	22%	4%	4%	0%
	VFS [4]	28	50%	4%	14%	25%	7%
	procfs [55]	18	44%	22%	22%	6%	6%
	ReiserFS [5]	5	60%	0%	40%	0%	0%
	Ext3 [80]	11	82%	9%	0%	9%	0%
	tmpfs [78]	6	33%	17%	50%	0%	0%
	Ext4 [52]	45	87%	4%	7%	2%	0%
	Btrfs [68]	13	54%	15%	8%	23%	0%
	F2FS [44]	18	94%	0%	6%	0%	0%
Mobile	HFS/HFS+ [38]	8	25%	13%	62%	0%	0%
	eCryptFS [36]	16	25%	25%	38%	0%	13%
	F2FS [44]	11	18%	55%	27%	0%	0%
	APFS [12]	4	0%	25%	75%	0%	0%
	Others	18	28%	17%	33%	11%	11%
Networked	NFS [63]	72	68%	1%	12%	18%	0%
	GlusterFS [29]	18	28%	17%	17%	38%	0%
	OpenAFS [3]	27	63%	30%	4%	4%	0%
	HDFS [37]	6	33%	17%	33%	17%	0%
	CephFS [84]	19	73%	20%	7%	0%	0%

D: DoS, L: Data Leakage, E: Privilege Escalation, A: Access Permission Bypass, O: Other (e.g., write access to `proc` entries like `CVE-2004-2613`).

Table 5. Correlation Between the File System Vulnerability Types and Their Threats

Type	#CVEs	D	L	E	A	O
Concurrency Issues	22	76%	8%	8%	8%	0%
Memory Errors	82	59%	26%	12%	1%	3%
Sanity Check Errors	111	60%	8%	18%	13%	1%
Permission Errors	56	20%	15%	26%	39%	0%
Network Errors	36	57%	16%	12%	10%	6%

We use the same taxonomy for vulnerabilities as in Table 2.

memory, where sensitive information, such as user data, file system metadata, and kernel memory unintentionally propagates to unauthorized destinations (e.g., `CVE-2005-0400` and `CVE-2004-0177`). *In comparison with other software systems, uninitialized memory in file systems can more often lead to data leakage.* As we pointed out in Section 3, file systems incorporate many cache systems to improve their performance. Memory objects used by the cache systems shall be flushed to the disk, which creates extra paths for uninitialized memory to leak.

The second type is permission-bypassing vulnerabilities. By exploiting such vulnerabilities, the adversaries can gain access to files without the proper authorization and, thus, indirectly leak private data. For instance, `CVE-2012-4508` allows adversaries to obtain sensitive information from a deleted file by exploiting a race condition to gain permissions.

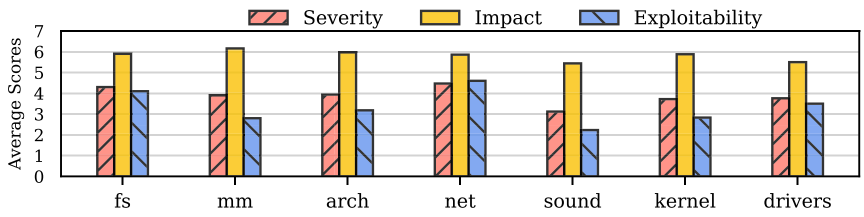


Fig. 6. Vulnerabilities of different core OS components. We measure the vulnerabilities in three dimensions: `Severity`, `Impact`, and `Exploitability`. For each CVE report, we obtain its severity score, CIA impacts (impacts of confidentiality-integrity-availability), and access complexity from CVSS (common vulnerability scoring system) [59]. Each CVSS score ranges from 0 to 10; the score of 10 represents the most severe, the most impactful, and the most exploitable vulnerability.

The last type is vulnerabilities in mobile file systems that enable physical attacks. The most direct consequence of such attacks is the breach of private data. For instance, physical adversaries, by exploiting `CVE-2014-7951` in Android Debug Bridge of Android 4.0.4, can gain a direct connection to the device to read/write arbitrary files in the file system.

We further study the severity of data leakage vulnerabilities in various types of file systems using the CVSS Severity Score [59]. We find that networked file systems have the highest severity for data leakage, with a Severity Score of 4.5. This was higher than the average scores for generic file systems (a score of 3.6) and mobile file systems (a score of 3.2). One possible reason for this is that networked file systems typically enable multi-tenancy. These vulnerabilities allow attackers to obtain password information from the server (`CVE-2020-10762, CVE-2020-10763`), potentially affecting all tenants of the networked file system.

`F-12`: Attackers can bypass access control via exploiting vulnerabilities in the enforcement of file permissions. Flaws in the enforcement of permissions largely become vulnerabilities. In comparison with other types of vulnerabilities, permission-related vulnerabilities can often be reliably exploited and consistently lead to permission bypassing.

Beyond the authentication of user identities, networked file systems also run access control on file access requests. Two categories of vulnerabilities in this type of access control can also be exploited to gain file permissions on the removed servers: (1) validation operations before setting ACL are missing; (2) permission checks are missing. For example, the NFS client in Linux 2.6.29.3 or before misses a check on the permission bits for execution, allowing local users to bypass permissions and execute files on an NFSv4 fileserver (`CVE-2009-1630`). Similar issues are also reported in `CVE-2005-3623` and `CVE-2016-1237`.

`F-13`: Adversaries can achieve privilege escalation via exploiting file system vulnerabilities. All the file systems we studied run at a high privilege. A significant subset of sanity-check vulnerabilities can enable adversaries to improperly "inherit" the high privilege. Some representative examples are: (i) `CVE-2017-5551` preserves the `setgid` bit during a `setxattr` call in a tmpfs file system, allowing local users to gain group privileges with restrictions on execute permissions; (ii) `CVE-2016-5393` wrongly gives HDFS service privileges to users who can only authenticate with the HDFS NameNode. This enables arbitrary commands execution with the same privileges as the HDFS service; (iii) `CVE-2016-1572` does not validate mount destination file system types, which allows local users to gain privileges by mounting over a nonstandard file system.

Authentication protocol used in modern networked file systems can also fail due to vulnerabilities and, thus, give users undeserved privileges. Current networked file systems either develop their own authentication protocol [84] or reuse the third-party infrastructure [62, 79]. Both ways of authentication can be tricked and bypassed. We observe a large number of such cases in Ceph and HDFS, mostly caused by mishandling of user credential messages (e.g., `CVE-2013-4134`, `CVE-2009-3516`).

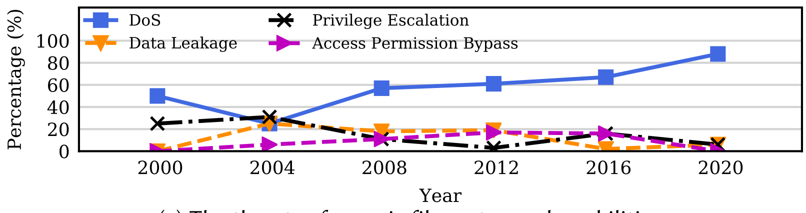
`F-14`: Vulnerabilities in mobile file systems can sabotage the protection of data privacy. To secure private data, mobile file systems have adopted protection mechanisms, mainly including encrypting user files [1] and isolating critical data inside trusted execution environments (TEEs). However, both mechanisms can be bypassed due to vulnerabilities. Since vulnerabilities related to file encryption have been discussed in Section 3 (see **`F-8`**), we now focus on TEEs below.

The TEE has become a standard feature on mobile devices, which has been leveraged by mobile file systems to enable an isolated environment for protecting sensitive data. However, a drop-in deployment of the TEE can be insufficient due to vulnerabilities in the interactions with the TEE. For instance, a system crash in the non-secure domain of a TrustZone-enabled ARM platform will trigger an interrupt to switch the system context to the secure domain. Therein, CPU, memory, and register states can be accessed. However, an armored adversary can prevent the non-secure domain to issue the context switch, and initiate a man-in-the-middle attack to obtain the execution information. For another example, eCryptFS uses an algorithm to generate a 32-byte AES key that combines the user password and 32-byte TIMA (TrustZone-based Integrity Measurement) key. The vulnerability (`CVE-2016-1919`) occurs because *Base64.getEncoder* expends the input with a ratio of 4:3. This means that only 24 bytes determine the final eCryptFS key and adversaries can easily crack the key with a brute-force search.

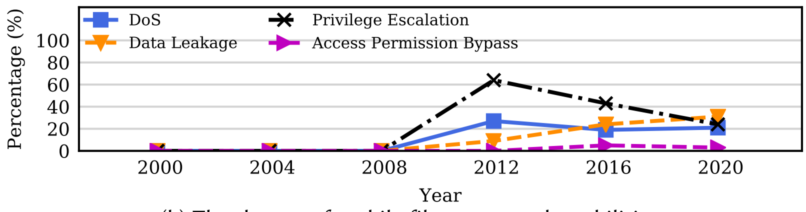
`F-15`: Protocol flaws in networked file systems can enable code injection attacks. Consider CephFS as an example. It uses the Ceph Object Gateway daemon (RGW) as an HTTP server to communicate with the Ceph storage cluster and retrieve file data. The HTTP response splitting (HRS) flaw (e.g., `CVE-2015-5245`) in the RGW can enable the well-known Carriage Return Line Feed (CRLF) injection and cross-site scripting (XSS) attacks. Once the malicious code is injected, attackers will have the privilege to compromise the victim server and then tamper the entire networked storage cluster.

`F-16`: Vulnerabilities in a single instance of networked file system can impose threats to the entire system. Networked file systems (e.g., HDFS [74] and GFS [28]) can often run a group of parallel instances on a cluster of servers. Each instance runs atop the local file system and delegates the file system to store data on the disk. The consequence of vulnerabilities in a single instance can propagate to all other instances in the same group. Consider GlusterFS as an example. GlusterFS supports *brick multiplexing* to reduce memory consumption, which allows multiple compatible bricks to share the same process and manage their own data volumes. However, a *NULL* pointer dereference (`CVE-2018-10914`) triggered by a client request in one instance can interrupt the brick process. This will propagate to other multiplexed brick processes, crushing the entire GlusterFS. Other similar cases include `CVE-2012-4417`, `CVE-2019-15538`, and `CVE-2020-24394`.

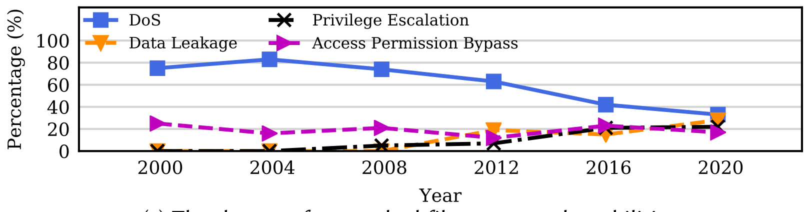
`F-17`: Vulnerabilities in different types of file systems present a different evolutionary trend in their threats. As shown in Figure 7, vulnerabilities in *generic* file systems tend to result in more DoS and fewer attacks of other types. However, vulnerabilities in *mobile* file systems and *networked* file systems present an opposite evolutionary trend. A presumed reason is that vulnerabilities in generic file systems are more conventional and their consequences are limited by mitigation mechanisms such as memory sanitizer [71] and thread sanitizer [72]. In contrast, many vulnerabilities in mobile/networked file systems belong to newly emerging types (e.g., protocol flaws) and have been less mitigated. Considering all types of file systems together, the evolution in the threat of their vulnerabilities presents no clear trend.



(a) The threats of generic file system vulnerabilities.



(b) The threats of mobile file system vulnerabilities.



(c) The threats of networked file system vulnerabilities.

Fig. 7. Evolutionary trend of the threats imposed by file system vulnerabilities. As for mobile file systems, no CVEs were reported before 2009.

Summary: The threats of file system vulnerabilities vary across different types, ranging from more functionality-critical ones such as DoS to more security-critical ones such as privilege escalation. Historical data shows that file system vulnerabilities, in general, are not becoming less harmful.
Insight-4: Our study unveils that different file system operations, when becoming vulnerable, can turn into different threats. This brings insights towards the mitigating process (e.g., patching): vulnerabilities involved in operations that can result in higher threats shall be prioritized. For instance, uninitialized memory errors in cache systems, which often incur data leakage, deserve an earlier fix than excessive memory consumption that typically incurs DoS. References about which file system operations can lead to what threats can be found from `F-10` to `F-16`.
Insight-5: Our study also unveils that vulnerabilities in mobile/networked file systems are presenting increasing threats (e.g., more data leakage than DoS). Considering the increasing popularity of mobile/networked file systems in the coming era of the IoT, our study brings evidence that more efforts should be prioritized to mitigate vulnerabilities in mobile/networked file systems.

5 PATCHING OF FILE SYSTEM VULNERABILITIES

Our taxonomy and threat analysis offer insights and motivations to mitigate file system vulnerabilities. We take a step further to analyze their patches and gain deeper insights. This will be beneficial to file system developers since it can help people learn about the common patterns

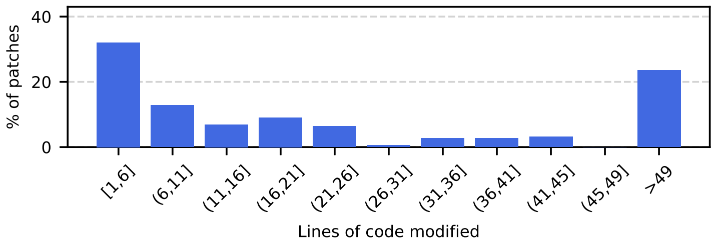


Fig. 8. Distribution of file system patches based on the lines of code (LoC) changed.

of patching file systems and guide them to avoid the deficiencies in today's practice. As stated in Section 2, we collected the open-sourced patches from CVE reports and obtained information such as the lines of code (LoC) added, LoC removed, and the commit time. Our study on the commit messages of each patch and the Debian Security Bug Tracker records [24] also shows that it takes 3.1 years on average from when the vulnerability was introduced to it being fixed, which is similar to other Linux system software [10]. Therefore, we do not study this further.

In the patch analysis process, we focus on answering these three questions: (1) *How complex are the patches?* (2) *How are the patches applied, systematically or in an ad-hoc manner?* (3) *Are patches across different file systems correlated and how may that affect the patching process?*

F-18: Most of the patches (56%) produce small modifications to the file systems. Figure 8 shows the distribution of patches based on their complexities (i.e., how many lines of code are modified). We find that a large number of patches are simply adding a check to the memory boundary (e.g., CVE-2013-6382) or fixing a syntax error (e.g., CVE-2013-1848). These patches constitute the low-complexity group. Most of the strategies are conventional, such as adding boundary checks to fix buffer overflows (e.g., CVE-2008-3531). For instance, the patches of many race conditions simply make the responsible operations atomic. The drawback of such ad-hoc strategies is they are too passive. They can neither help prevent similar vulnerabilities in the codebase nor bring global benefits, such as new designs. We believe that a more systematic strategy should be taken (see our discussion in **F-20**).

F-19: A non-negligible portion (20%) of the patches involve complex modifications, such as fixes to multiple file systems and comprehensive validations of file system data structures. Some vulnerabilities that are shared across file systems can cause duplicated patches. Most of the patches we studied aim to solve a unique flaw in a certain file system. In the study, we discovered that for a shared vulnerability in multiple file systems, developers release patches that have a similar strategy or even the same code logic. These patches build up the high-complexity group. Strategies of a certain patch to a specific file system can be borrowed by patches that are used by other types of file systems. For example, as shown in Figure 9(a), the VFS subsystem added a new function setattr_killpriv to remove extended privilege attributes (xattrs). Since most Linux file systems support xattrs, they have to properly invoke this function after the permission check *inode_change_ok*() to ensure security. As a result, these file systems generate similar patches (Figures 9(b) and 9(c), CVE-2015-1350). This pattern also occurs in the patching of xattr block caching in the ext2 and ext4 file systems. Both of the file systems have to convert the original meta block cache from mbcache to mbcache2. Hence, they should add the same sanity checking to the entries and generate similar patches (e.g., CVE-2015-8952).

A cross-file system patching approach could offer two major benefits. First, having a unified solution simplifies the patch development process as well as the future maintenance. Second, file

```
int setattr_killpriv(struct dentry *dentry, struct iattr *iattr)
  if (!(iattr->ia_valid & ATTR_KILL_PRIV))
   return 0;
   iattr->ia_valid &= ~ATTR_KILL_PRIV;
   return security_inode_killpriv(dentry);
```

(a) A new interface added to VFS that removes extended privilege attributes.

```
int ext3_setattr(struct dentry *dentry,
     struct iattr *attr)
  error = inode_change_ok(inode, attr);
  if (error)
    return error;
+ error = setattr_killpriv(dentry, attr);
+ if (error)
+ return error;
```

(b) Patch to ext3 filesystem.

```
int btrfs_setattr(struct dentry *dentry,
     struct iattr *attr)
  error = inode_change_ok(inode, attr);
  if (err)
    return err;
+ err = setattr_killpriv(dentry, attr);
+ if (err)
+ return err;
```

(c) Patch to btrfs filesystem.

Fig. 9. Similar patches to different Linux file systems for fixing the same VFS vulnerability.

systems that share the patch can take advantage of further improvements in each individual file system, which enables us to improve the overall robustness and reliability.

F-20: Some patches demonstrate systematic strategies, which can benefit vulnerability prevention. Despite ad-hoc strategies still being dominant, we do observe systematic strategies in two cases. First, there are patches that perform systematic verification on file system data structures instead of simply applying an ad-hoc fix. For instance, CVE-2018-14613 reports a simple invalid pointer dereference issue when mounting and operating a crafted btrfs image caused by invalid block group items. This specific vulnerability could have been solved by a low-complexity patch. However, the patch to it demonstrates good practice by adding a new sanity check function that performs a complete check of the item size, offset, object id, type, and used space. This not only helps fix the reported vulnerabilities but also removes the undiscovered hidden flaws to prevent future vulnerabilities. Indeed, none of the follow-up CVEs has related issues when mounting and operating on btrfs images. Second, many patches of race conditions switch from lock-based implementations to lock-free implementations, such as deploying semaphore-based solutions (e.g., CVE-2014-9710) or avoiding race conditions in the first place (e.g., CVE-2014-8086). These patches bring design-wise insights, which are applicable to a more fundamental resolution of race conditions.

F-21: Patches that modify the VFS need coordinated updates to the underlying file systems. A patch to the VFS subsystem often requires updating the underlying file systems. Otherwise, the patch may not come into effect. In CVE-2015-1350, the VFS subsystem provides an incomplete set of requirements for setattr operations, which under-specifies the removal of extended privilege attributes. The corresponding patch silently introduces a new VFS API to fix the flaw. Concurrently, the patch has to modify all 21 affected underlying file systems to use the new API. In this example, the complexity incurred by the need for coordinated patching is well handled, but this is not always the case. For example, the patch to CVE-2016-7097 modifies VFS but fails to update the complete set of 15 affected underlying file systems, leaving the vulnerability still open (reported as CVE-2017-5551). Until one year later, another patch to CVE-2017-5551 eventually fixed the problem.

Table 6. Suggested Strategies to Patch Common Types of File System Vulnerabilities

Type	Suggested Defense Strategies	#CVEs	% of Applicable CVEs
Sanity checks	FS image validation before mounting	53	47.7%
	Validating fs parameters upon file access	14	12.6%
	Memory pointer validation and bounds checking	6	5.4%
	Handle exceptions of division by zero	4	3.6%
	Avoid integer overflows	4	3.6%
Concurrency issues	Enforce atomic ToCToU race operations	9	40.9%
	Support of lock-free optimizations	6	27.3%
Memory errors	Initialize unused, allocated memory	32	39.0%
	Initialize data before copying them from user space	20	24.4%
	Reinitialize fs metadata upon file updates	12	14.6%
File permissions	Check enforcement of user privileges	24	42.9%
	Check file permissions for direct storage access	16	28.6%
	Validate fs metadata information before mounting fs	3	5.4%

The third column shows the percentage of CVEs that can be fixed with the suggested strategies.

Summary: Patches to file system vulnerabilities carry different complexities. The patches also adopt various strategies. Most of them are ad-hoc, bringing no help with addressing similar but yet unknown vulnerabilities. However, patches following systematic strategies are arising. These patches present preventive measures or new designs, offering insights towards fundamentally mitigating vulnerabilities of the same root causes. Finally, file systems are often correlated. However, today's practice often overlooks the correlated file systems when patching the target ones, thus, leading to insecure patches.
Insight-6: Our study unveils that the current practice of patching file systems is largely passive. Continuing this practice will likely prolong the arms race between vulnerability creation and patching. To fundamentally escalate the security of file systems, the community should encourage and promote the adoption of preventive and secure-by-design patching. Exemplary strategies observed by our study have been presented in `F-20`.
Insight-7: Better attention needs to be paid to the correlation of file systems when developing patches. To ensure the correctness of patches, any modification to a file system (in particular, VFS) must be validated to understand its impacts on other file systems. Whenever such impacts arise, the affected file systems must be updated accordingly.

6 SUGGESTIONS

In this study, we find that a large portion of file system vulnerabilities are semantic related or use-context related. We summarize the common suggested strategies used by the patches on generic file system vulnerabilities in Table 6. The common defense strategies discussed of patching generic file systems can be also applied to fix 44% of the mobile file system vulnerabilities and 34% of the networked file system vulnerabilities. Here, we identify their unique defense strategies' vulnerabilities, as shown in Tables 7 and 8.

`S-1`: Verifying file system implementations with semantics specified in VFS. As discussed, VFS provides a uniform abstraction level for upper-level programs to interact with real file system implementations. It offers semantics (e.g., POSIX) for accessing the underlying file systems. However, the real file system implementation may not exactly satisfy the semantics due to the unclear specifications between the VFS and low-level file system implementations (e.g., `CVE-2015-1350` and `CVE-2016-7097`). Prior verification studies of file systems [57, 75] mostly worked on crash safety. Our findings reveal that more efforts are required to achieve the end-to-end verification

Table 7. Suggested Defense Strategies for Mobile *fs* Vulnerabilities

Suggested Defense Strategies	# CVEs	% of Applicable CVEs
Enhance authentication mechanisms in ADB channels	14	24.5%
Enhance USB debugging mode	11	19.3%
Enforce secure interaction between TrustZone and FS	7	12.3%
(Total)	32	56.1%

The second column shows the percentage of CVEs that could be fixed with the suggested strategies.

Table 8. Suggested Defense Strategies for Networked *fs* Vulnerabilities

Suggested Defense Strategies	#CVEs	% of Applicable CVEs
Enforce strict authentication/permission control	37	26.1%
Verify (HTTP/RPC) network packets	33	23.2%
Enforce VFS specifications	13	9.2%
Fault isolation between *fs* instances	10	7.0%
(Total)	93	65.5%

The second column shows the percentage of CVEs that could be fixed with the suggested strategies.

of each semantic mapping from the high-level VFS specification to the low-level file system implementation.

S-2: Integrity check for file system images. Attackers can exploit the flaws in sanity checks to bypass the validation of illegal file system data to initiate their attacks. Since the file system provides data durability, the transient errors or modifications to the in-memory data structures could be persisted to the disk (file system image), causing long-term threats to the host system. Moreover, attackers can leverage crafted file system images with incorrect attributes, invalid links, or illegal data to compromise the file system or the entire operating system (e.g., CVE-2018-14613 and CVE-2016-1572). To defend against such attacks, we encourage preventive measures or new designs to enforce integrity checks on file system images before mounting.

S-3: Enhancing secure channels between mobiles and external physical devices. Due to the unique attack surface of mobile file systems, they need to enforce stricter security checks against physical attacks. As described in Section 3 and F-8, attackers could exploit flaws in encryption mechanisms in mobile file systems. We need to further develop secure channels between mobile devices and external devices such as Thunderbolt adapters and USB ports. According to our study, nearly 23% of vulnerabilities in Android can be exploited by evil-maid attacks. These attacks can be prevented by explicitly authorizing the mounting of external devices.

S-4: Enforcing secure communication between TEE and file systems. We still cannot solely rely on TEE or encrypted file systems or even both to ensure the security of mobile file systems. This is because the file system might not be able to correctly use, interact with, and manage the trusted environments (e.g., CVE-2017-13786), and the encryption itself might be compromised by brute force attacks (e.g., CVE-2016-1919). In particular, we have to ensure the secure interactions between TEE and file systems, such as preventing the direct access to the memory used by the Volume Daemon or storage management process from TrustZone. We have to carefully refine the interfaces between them, which can be improved with the development at both the TEE and file system sides [20, 65, 87].

S-5: Verifying network packet. According to our study, verifying the network packets, especially those via the HTTP or RPC protocols, can significantly reduce the risk of network attacks in networked file systems. The semantic specifications from upper-level service protocols defined by network file systems will facilitate such network packet verification. Similar network verification techniques have been developed in software-defined networking domains [66, 67] and packet level authentication [47]. However, few of them focused on the semantic-aware verification of storage traffic from networked file systems.

S-6: Fault isolation for networked file system instances. As discussed in Section 4, a file system instance (e.g., a data volume manager) has correlations with many system components, such as local file systems and OS kernel. These correlations significantly complicate the management of file system instances and increase the attack surface. Providing a fault isolation mechanism for file system instances can significantly enhance the security of the entire networked system. Systems techniques, such as containers [46, 73], sandbox [17], and TEE [13], have been proposed to enforce the isolation between programs. They can be leveraged to enable isolation between file system instances. However, a holistic approach that requires fine-grained function partition and placement for networked file systems is still highly desirable [94].

7 RELATED WORK

Study of Bugs and Vulnerabilities. Past research has conducted many studies on defects in operating systems. The studies vary in targets. Many of them [49, 54, 83, 88] focus on functionality bugs in file systems while some others [30, 40] concentrate on virtual memory management problems. The studies also vary in scope that ranges from concurrent issues [26, 34, 45, 51] to specific families of vulnerabilities [22, 48, 70]. Our work uses a methodology similar to many of the studies. However, we focus on the security vulnerabilities in file systems. Unlike the previous file system studies that focused on bug causes and consequences [49], our study centers around the understanding of the attack surface of file systems and the vulnerability exploitation procedure, while covering a wider range of CVEs across different types of file systems. To the best of our knowledge, our study is the first in-depth work of its kind.

Bug and Vulnerability Detection. Research in this line can be classified into two categories: bug finding and formal verification. Bug finding tools, such as FiSC [91], eXplode [90], and juxta [54], can discover file system bugs based on semantic-aware patterns. Our study can benefit these tools by providing insights into expanding their patterns. Formal verification [11, 15, 21, 23, 32, 57, 75] is also promising. However, at this stage, it is still challenging to verify the security of an entire file system due to the high complexity [32, 75, 86]. Our study can complement formal verification. It pinpoints the file system components that are more vulnerable, thus enabling formal verification to narrow down the scope and scale up.

Secure File Systems. Besides bug study and detection, researchers have also been endeavoring to build secure and reliable file systems [50, 53]. Lu et al. [50] proposed the physical disentanglement to minimize storage faults propagation. Min et al. [53] leveraged transaction flash storage to build crash-consistent file systems. Our study can bring insights to follow-up works in this line of research. For instance, we find that many vulnerabilities in networked file systems are caused by the lack of fault isolation across different instances. This can motivate efforts to develop fault isolation mechanisms for file systems.

8 CONCLUSION

This article presents an empirical study on the security of modern file systems, following the angle of inspecting vulnerabilities disclosed from mainstream file systems. Throughout the study, we build the first systematic understanding of the attack surfaces faced by file systems, the threats tied to the attack surfaces, and the limitations in today's practice of addressing the attack surfaces.

We envision that our study will raise awareness of file system security and, more importantly, offer insights towards improving file system security.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments and feedback.

REFERENCES

[1] n.d. Apple File System secure encryption mechanism. https://9to5mac.com/2016/06/13/apple-file-system-apfs/
[2] n.d. Journaled File System Technology for Linux. http://jfs.sourceforge.net
[3] n.d. OpenAFS. https://www.openafs.org
[4] n.d. Overview of the Linux Virtual File System. https://www.kernel.org/doc/html/latest/filesystems/vfs.html
[5] n.d. ReiserFS. https://en.wikipedia.org/wiki/ReiserFS
[6] 2011. Metadata Replication for Ext4. https://lwn.net/Articles/465041/
[7] 2021. Request CVE IDs. https://cve.mitre.org/cve/request_id.html
[8] Hiralal Agrawal, Joseph Robert Horgan, Edward W. Krauser, and Saul A. London. 1993. Incremental regression testing. In *1993 Conference on Software Maintenance.* IEEE, 348–357.
[9] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *Network and Distributed System Security Symposium, (NDSS'18).* San Diego, CA, USA.
[10] Nikolaos Alexopoulos, Manuel Brack, Jan Wagner, Tim Grube, Max Mühlhäuser, Andrew Meneely, and Dorian Arnouts, Emmanouil Vasilomanolakis, Stephane Le, Steven Rowe, et al. 2022. How long do vulnerabilities live in the code? A large-scale empirical measurement study on FOSS vulnerability lifetimes. In *31st USENIX Security Symposium (USENIX Security'22).* 359–376.
[11] 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 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16).* Atlanta, GA.
[12] n.d. APFS: Apple File System. https://en.wikipedia.org/wiki/Apple_File_System
[13] Pierre-Louis Aublin, Florian Kelbert, Dan O'Keeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2018. LibSEAL: Revealing service integrity violations using trusted execution. In *Proceedings of the 13th EuroSys Conference (EuroSys'18).*
[14] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2009. Tolerating file-system mistakes with EnvyFS. In *USENIX Annual Technical Conference*, Vol. 9.
[15] 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 21st International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'16.* Atlanta, GA, USA.
[16] Miao Cai, Hao Huang, and Jian Huang. 2019. Understanding security vulnerabilities in file systems. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'19).* Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3343737.3343753
[17] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. 2010. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10).*
[18] Remy Card. 1995. Design and implementation of the second extended filesystem. In *Proceedings of the First Dutch International Symposium on Linux, 1995.*
[19] n.d. Ceph. https://en.wikipedia.org/wiki/Ceph_(software)
[20] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland'20).*
[21] 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 (SOSP'17).* Shanghai, China.
[22] 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.
[23] 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.
[24] Debian. 2022. Security Bug Tracker. https://security-tracker.debian.org/tracker/

[25] 2020. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Flexible_Block_Groups

[26] 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.

[27] Andy Galloway, Gerald Lüttgen, Jan Tobias Mühlberg, and Radu I. Siminiceanu. 2009. Model-checking the Linux virtual file system. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 74–88.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP'03)*. 29–43.

[29] GlusterFS. n.d. GlusterFS architecture. https://docs.gluster.org/en/v3/Quick-Start-Guide/Architecture/

[30] Sudhakar Govindavajhala and Andrew W. Appel. 2003. Using memory errors to attack a virtual machine. In *2003 Symposium on Security and Privacy (SP)*. Berkeley, CA, USA.

[31] Andreas Gruenbacher. 2003. POSIX access control lists on Linux. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC'03)*. San Antonio, TX.

[32] 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.

[33] 2019. Researcher Reservation Guidelines. https://cve.mitre.org/cve/researcher_reservation_guidelines

[34] 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, SoCC 2014*. Seattle, WA, USA.

[35] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2007. Improving file system reliability with i/o shepherding. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. 293–306.

[36] Michael Austin Halcrow. 2005. eCryptfs: An enterprise-class encrypted filesystem for Linux. In *Proceedings of the 2005 Linux Symposium*, Vol. 1. 201–218.

[37] 2019. Haoop Distributed File System. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[38] n.d. HFS: Hierarchical File System. https://en.wikipedia.org/wiki/Hierarchical_File_System

[39] 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, CA, USA.

[40] Jian Huang, Moinuddin K. Qureshi, and Karsten Schwan. 2016. An evolutionary study of Linux memory management for fun and profit. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. Denver, CO, USA.

[41] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. 2018. Designing a true direct-access file system with DevFS. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*. Oakland, CA.

[42] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2017. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)*. Ontario, Canada.

[43] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. 2021. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. 75–92.

[44] 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.

[45] 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 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. Atlanta, GA, USA.

[46] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. 2018. A measurement study on Linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC'18)*.

[47] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical verification for programmable data planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*.

[48] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. 2017. Unleashing use-before-initialization vulnerabilities in the Linux kernel using targeted stack spraying. In *Network and Distributed System Security Symposium, (NDSS'17)*. San Diego, CA, USA.

[49] 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'13)*. San Jose, CA.

[50] 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.

[51] 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.

[52] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2007. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*.

[53] 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*. Santa Clara, CA, USA.

[54] 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.

[55] Erik Mouw. 2001. Linux kernel procfs guide. *Delft University of Technology* (2001).

[56] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 919–936.

[57] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*.

[58] 2020. Network File System. https://en.wikipedia.org/wiki/Network_File_System

[59] n.d. NVD. https://nvd.nist.gov/vuln-metrics/cvss

[60] National Institute of Standards and Technology. 2022. Attack Surface. https://csrc.nist.gov/glossary/term/attack_surface

[61] Akira K. Onoma, Wei-Tek Tsai, Mustafa Poonawala, and Hiroshi Suganuma. 1998. Regression testing in an industrial environment. *Commun. ACM* 41, 5 (1998), 81–86.

[62] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. 1994. NFS version 3: Design and implementation.. In *USENIX Summer Technical Conference*. Boston, MA, USA.

[63] Brian Pawlowski, David Noveck, David Robinson, and Robert Thurlow. 2000. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*. Maastricht, Netherlands.

[64] 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.

[65] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A comprehensive survey. *ACM Comput. Surv.* 51, 6 (Jan. 2019).

[66] Gordon D. Plotkin, Nikolaj Bjørner, Nuno P. Lopes, Andrey Rybalchenko, and George Varghese. 2016. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*.

[67] Santhosh Prabhu, Kuan Yen Chou, Ali Kheradmand, Brighten Godfrey, and Matthew Caesar. 2020. Plankton: Scalable network configuration verification through model checking. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*. Santa Clara, CA.

[68] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)* 9, 3 (2013), 1–32.

[69] Mendel Rosenblum and John K. Ousterhout. 1991. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating System Principles, SOSP 1991*. Pacific Grove, CA, USA.

[70] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-assisted feedback fuzzing for {OS} kernels. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*. San Jose, CA, USA.

[71] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[72] Konstantin Serebryany and Timur Iskhodzhanov. 2009. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*. 62–71.

[73] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking down barriers to improve performance and isolation of cloud-native

containers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[74] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. 2010. The Hadoop distributed file system.. In *IEEE 26th Symposium on Mass Storage Systems and Technologies, (MSST'10)*. Incline Village, NV, USA.

[75] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA, USA.

[76] Satyam Singh. 2021. Exploiting NFS Share. https://resources.infosecinstitute.com

[77] Stephen Smalley, Chris Vance, and Wayne Salamon. 2001. Implementing Selinux as a Linux security module. *NAI Labs Report* 1, 43 (2001), 139.

[78] Peter Snyder. 1990. tmpfs: A virtual memory file system. In *Proceedings of the Autumn 1990 EUUG Conference*. 241–248.

[79] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. 1988. Kerberos: An authentication service for open network systems.. In *USENIX Winter Technical Conference*. Dallas, TX, USA.

[80] Stephen Tweedie. 1998. Journaling the Linux ext2fs filesystem. In *4th Annual Linux Expo, 1998*. Durhan, NC, USA.

[81] 2019. VulDB. https://vuldb.com/?id.141577

[82] Randolph Y. Wang and Thomas E. Anderson. 1993. xFS: A wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*. IEEE, 71–78.

[83] Jinpeng Wei and Calton Pu. 2005. TOCTTOU vulnerabilities in Unix-style file systems: An anatomical study. In *4th USENIX Conference on File and Storage Technologies (FAST'05)*. San Francisco, CA, USA.

[84] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating systems Design and Implementation (OSDI'06)*. Seattle, WA, USA.

[85] Wikipedia contributors. 2022. Attack surface - Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Attack_surface

[86] 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.

[87] Meng Xu, Chengyu Song, Yang Ji, Ming-Wei Shih, Kangjie Lu, Cong Zheng, Ruian Duan, Yeongjin Jang, Byoungyoung Lee, Chenxiong Qian, Sangho Lee, and Taesoo Kim. 2016. Toward engineering a secure Android ecosystem: A survey of existing techniques. *ACM Comput. Surv.* 49, 2 (Aug. 2016).

[88] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy (Oakland'19)*. San Francisco, CA, USA.

[89] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[90] Junfeng Yang, Can Sar, and Dawson R. Engler. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*. Seattle, WA, USA.

[91] 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, CA, USA.

[92] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[93] 2022. ZDNet. https://www.zdnet.com/article/fbi-warning-this-ransomware-gang-has-hit-over-100-targets-and-made-more-than-60-million/

[94] Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford. 2014. Heading off correlated failures through independence-as-a-service. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO, USA.

Received 17 June 2022; revised 31 December 2022; accepted 17 March 2023