Journal of *Cybersecurity and Privacy*

MDPI

Article

Reversing File Access Control Using Disk Forensics on Low-Level Flash Memory

Caleb Rother and Bo Chen *

Department of Computer Science, Michigan Technological University, Houghton, MI 49931, USA; crother@mtu.edu

* Correspondence: bchen@mtu.edu; Tel.: +1-906-487-3149

Abstract: In the history of access control, nearly every system designed has relied on the operating system (OS) to enforce the access control protocols. However, if the OS (and specifically root access) is compromised, there are few if any solutions that can get users back into their system efficiently. In this work, we have proposed a novel approach that allows secure and efficient rollback of file access control after an adversary compromises the OS and corrupts the access control metadata. Our key observation is that the underlying flash memory typically performs out-of-place updates. Taking advantage of this unique feature, we can extract the "stale data" specific for OS access control, by performing low-level disk forensics over the raw flash memory. This allows efficiently rolling back the OS access control to a state pre-dating the compromise. To justify the feasibility of the proposed approach, we have implemented it in a computing device using file system EXT2/EXT3 and open-sourced flash memory firmware OpenNFM. We also evaluated the potential impact of our design on the original system. Experimental results indicate that the performance of the affected drive is not significantly impacted.

Keywords: access control; recovery; disk forensics; flash memory; file system

check for updates

Citation: Rother, C.; Chen, B. Reversing File Access Control Using Disk Forensics on Low-Level Flash Memory. *J. Cybersecur. Priv.* **2024**, *1*, 1–18. https://doi.org/

Academic Editor: Firstname Lastname

Received: 14 August 2024
Revised: 19 September 2024
Accepted:
Published:

1. Introduction

Access control is a critical part of the metadata of any file. Be it Linux's iconic 777-style permissions [1] or Windows's read-only checkboxes, each operating system needs some method of determining who can perform operations on a file. To ensure that the metadata can work correctly for controlling access of system resources, an implied assumption is that the OS should not be compromised. This assumption however, does not always hold. Major commodity operating systems use large monolithic kernels which are prone to attacks [2–4]. Once the OS is compromised, the adversary may arbitrarily change the file permissions by modifying desired metadata through commands like chmod [5]. This allows for easy creation of backdoor and leaked data, both of which are major threats to the security of any organization. After the adversary is detected and evicted, efficiently restoring file permissions is a crucial task in timely returning the system to its normal state. However, the malicious changes to the metadata could be wide-scale and efficiently restoring the state of files is a non-trivial problem. The core of the issue is a lack of ability to track and restore metadata across a drive, specifically metadata pertaining to access control/file restrictions. In the scenario of an attacker compromising the OS, a backdoor created by changing file permissions could persist past the removal of the attacker themselves. In order to remove this type of backdoor, changes to access control data need to be reverted. In this work, we aim to design such a system, capable of identifying generalized changes to the access control metadata and reversing them accordingly. Our design takes advantage of the out-of-place update strategy, which is implemented internally in flash memory storage media that has become dominant in the modern computer storage market.

There are several key aspects of filesystems and flash memory which our design relies on: (1) Filesystems must write all metadata to the drive somehow, a command that must

be handled by a lower layer. This means that if we were able to write a program that uses that lower layer, we would be able to track metadata writes. (2) Most flash storage drives like SSDs introduce a flash translation layer (FTL), a firmware layer staying between the OS and the underlying NAND flash, which can transparently manage the unique hardware nature of raw flash, exposing a block access interface externally. This firmware layer yields a solution to the problem posed previously, giving us an area below the native operating system which can run our customized programs. (3) Part of the standard kit of flash memory is a set of small "out of bounds" (OOB) areas which can be used to store miscellaneous data [6]. This may be usable in order to prevent large-scale memory use in sections of code which must be fast in order to properly write data. (4) FTL utilizes a strategy called out-of-place updates to perform write operations which leaves old copies of data on the drive [7]. These copies of data can linger for extended periods of time, and this time can also be extended by modifying the code used for the update procedure. This allows us to keep older versions of data on the drive for an extended period of time.

These areas combine to allow for a system which tracks metadata writes and stores the previous location in an OOB area. After this is stored, the user can press some kind of manual override switch upon detection of an intrusion and reset any recent changes to access control metadata that they may wish to. There are, however, technical difficulties. Filesystem forensics must be conducted within the FTL, necessitating strategies to minimize memory usage. In addition, performing the forensics can be difficult due to the lack of OS support. We need to manually decode the raw binary data on the drive without relying on any built-in OS functions designed for this purpose. There is documentation for this, but navigation must still be done manually by interpreting the data. We accomplish this by following relevant chains of binary data and only saving the starts and lengths of groups of data which we want to track. This allows us to make a memory efficient system without sacrificing significant amounts of performance.

Contributions. Major contributions of this work are listed below:

- We have developed ACRecovery, the pioneering access control recovery scheme capable of restoring access control metadata after the eviction of an adversary that has compromised the operating system.
- ACRecovery ensures recoverability of the access control metadata in the OS by (1) leveraging the out-of-place update implementation in the FTL, and (2) performing low-level disk forensics over the raw NAND flash.
- We have implemented a prototype of ACRecovery when deploying EXT2/EXT3 as the filesystem. Experimental evaluation demonstrates that ACRecovery can efficiently roll back the access control metadata, with a small impact on the performance of the equipped flash memory storage.

2. General Background

2.1. File Metadata

File metadata is any data which pertains to a file but is not directly part of it. This can include authorship information, modification time, creation time, and more. One particularly relevant section of metadata is permissions data, which can be represented in several ways. In EXT filesystems, for instance, permissions are represented as a series of 12 bits. The first three bits cover read, write, and execution permissions for the owner of the file, the second set of three handles the same permissions for the owner's group, and the third set for all other users.

2.2. Filesystem

A filesystem takes OS instructions for file creation/deletion and settings changes and translates them to read/write instructions to deliver to the drive the OS is stored on [8]. Filesystems are responsible for almost every binary representation of data coming from the OS. Typically, filesystems store data in the forms of either files or nodes, depending on what they represent. Other forms of data storage exist, but there must be some method

of finding any single section of data relatively quickly due to OS demands. The objects represented include file authors, related groups, tracking metadata such as size and access permissions, directories and subdirectories, etc. Since the filesystem must store all of this data, anything that can access and interpret the data on the filesytem can read the drive as though it were the OS. We leverage this possibility to recover access control data.

EXT file systems. EXTended File System (EXT) was made for Linux in the 1990s and is still used as its primary filesystem in the modern age. The system has gone through several iterations, i.e., EXT2 [9], EXT3 [10], and EXT4 [11]. It is based around a series of records for boot/basic controls, then uses a series of "inodes" to manage data. These inodes store everything from access control schemes to file size to creation time. In general, any non-direct data relating to a file is held in the inodes. These inodes are stored according to a series of groups on the drive. The inodes for a single group are stored contiguously, but the groups are not stored back-to-back. File data is stored across the drive, with inodes containing pointers to it. All of the data relating to these groups (but not the files they contain) is stored in a "superblock", a set of data regarding the filesystem stored at the beginning of the drive. Figure 1 demonstrates the structure of an EXT filesystem.

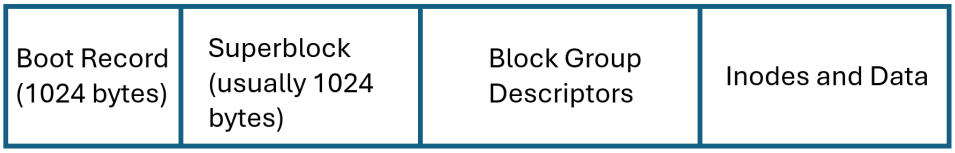

Figure 1. A general layout of an EXT partition.

Notice that the superblock is stored at the beginning of the partition, followed by a series of "block group descriptors", followed by data intermixed with "inode groups". The superblock stores a wide variety of data related to the filesystem, including the sizes and numbers of various data structures, version data, and which inodes are allocated. Immediately following the superblock is a set of "block group descriptors". These dictate data revolving around inode groups and their locations. This stores the number of inodes, the start location of the set of inodes for the group, and the number of directories the group contains [9,12].

In 1999, EXT3 was introduced in order to add support for a technique called journaling. This is a method of tracking writes to a drive which have not yet been committed. This is the main included feature of EXT3 [10]. This version of the filesystem is still used for legacy applications and can be found on some modern systems, giving us a more up-to-date filesystem to test on.

2.3. Master Boot Record

The Master Boot Record (MBR) of a drive stores basic information about what is contained over the entire drive. This includes a list of the filesystems on the drive and which sections they have charge over, called partitions. The MBR is always at the very beginning of the drive, and as such any attempt at filesystem forensics needs to read it first. Importantly, the bytes corresponding to partition data are stored between bytes 0x1BE and 0x1FD of the drive [13].

2.4. Access Control List

An access control list (ACL) is a list of permissions associated with a system object. It has been implemented in a variety of file systems to control access of files and directories, including but not limited to NTFS (used in Windows), ext2/ext3/ext4 (used in Linux and other Unix-like OSes), APFS (used in macOS and iOS), etc. [14,15]. These methods include:

1. Having dedicated files to store the ACL and referencing that file when accessing others.
2. Spreading the ACL out over all the files' metadata, and reading the associated metadata when accessing the target file.

Regardless of how it is implemented, the ACL always contains some kind of permission and access data related to the files, directories, and objects on the system. We seek to ensure that this data stays intact, even through a complete corruption of the operating system.

2.5. Flash Memory

NAND flash has been used broadly as the external storage in both standard computers and mobile devices today [16], as it consumes much less power and is capable of much faster read/write speeds. Flash memory is typically organized into blocks, each of which consists of pages which contain a certain number of bytes of data. Each page contains a small out-of-bounds (OOB) area [6]. This OOB area is a small number of bytes (16 in our test drive) which is used to store extra data, such as error correcting codes. However, this OOB space is not typically fully used, and as such can be taken advantage of. Flash memory also has several unique hardware characteristics including erase-before-write, suffering from wear, etc. These unique features distinguish flash memory from traditional mechanical drives.

Erase-before-write. The way in which NAND flash performs writes is unique. Each page has a default state which is entirely made up of '1's. When a write is performed and a buffer is provided, the drive simply sets any '1's to '0's where the buffer directs, leaving other bits alone. As one would expect, a write operation run on the disk is performed at the smallest size the drive can handle, one page. However, because of this unique system, an erase operation is performed on the scale of an entire block of (usually 64) pages. As a result, an entire block must be erased before a page can be overwritten and erasures are treated as a very expensive operation [17].

2.6. Flash Translation Layer (FTL)

To remain compatible with traditional block file systems, a flash storage device (e.g., an SSD) usually exposes a block access interface. This is achieved by introducing a new flash translation layer (FTL), a piece of special firmware which stays between the file system and the NAND flash, transparently managing the special characteristics of NAND flash [18–20].

Figure 2. The layers used in operation of a filesystem and FTL.

Out-of-place updates. The FTL implements an out-of-place update strategy [7,21,22]. Since the erasure is so expensive, it is unfeasible to overwrite a page by actually writing the location the page is stored at again. As such, the FTL uses a system of logical and physical page addresses, linked by a page mapping table (PMT). The logical address of a page is where it appears to be to the OS and filesystem, and does not change as the page is overwritten. The physical address, on the other hand, is the actual location of the data on the drive. If this page is searched by the FTL, it will pull the data located exactly at that location regardless of any translation. The PMT handles translation between the two addresses, stylized as a table which maps the logical address of a page to its physical counterpart. In out-of-place updates, a write to a logical address instead causes a new physical page to be found which is currently empty. The new page is then written and the PMT changes the logical address of the target page accordingly. But, the data previously stored at the logical address is temporarily retained on the original physical page (*Note that our work mainly utilizes this phenomenon to restore the ACL entries compromised by the adversaries at the OS level*). In other words, given the logical address on the block device, by searching the PMT table, we can identify the physical flash page where the corresponding data is stored. An example of the PMT table is shown in Figure 3.

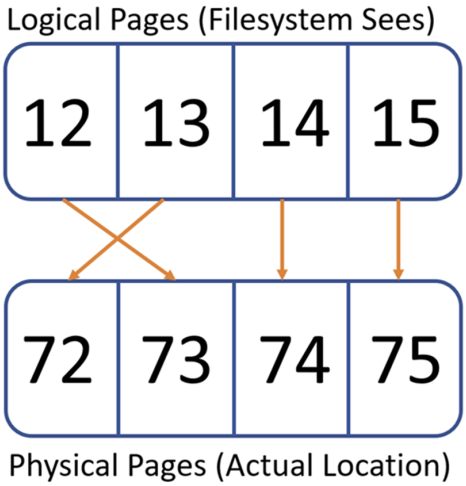

Figure 3. An example of the PMT table.

Wear leveling. Flash blocks have a limited number of erasures that they can perform before losing reliability. As the out-of-place update scheme is also implemented in flash systems, the best way to determine where data should be placed on the drive must be determined. This is done by a system called wear leveling, which periodically checks how many erasures have been performed on blocks across the drive and moves pages to other locations on the drive accordingly. This increases the lifespan of the drive as each block will take approximately equal erasure counts.

Garbage collection. A side effect of the out-of-place update and wear leveling systems is the fact that blocks of data will eventually fill up with out-of-date, or "dirty", pages. These blocks must be erased in order to use the full drive. The FTL periodically checks for blocks which have a certain number of dirty pages (or greater) and moves any valid data to a new block before erasing the old one. The PMT is then adjusted. This allows for filled blocks to be emptied and reused.

3. Related Work

Leveraging the FTL's out-of-place update implementation to restore the data compromised by either the ransomware or the malware has been studied extensively in the literature. However, none of them has been specifically designed for access control and cannot be used to restore the access control metadata.

Flash-based data recovery from ransomware attacks. There were various ransomware defending designs such as FlashGuard [23], MimosaFTL [24], and SSD-insider [25,26] which can restore the data compromised by the ransomware. A common technique used in those designs is to roll-back the entire invalidated raw flash memory data after the ransomware attacks, utilizing the FTL's out-of-place update strategy. Having observed that restoring the entire storage is time-consuming and may not be necessary, FFRecovery [27] allows the user to restore individual files compromised by the ransomware. This is achieved by additionally incorporating the filesystem forensics which can restore the filesystem metadata. By linking the restored filesystem metadata and the corresponding file data (restored by extracting the corresponding raw flash memory data guided by the filesystem metadata), a fine-grained recovery of user files is possible. Amoeba [28,29] is a method for backing up full hard-drives based on scanning the content and detecting similarities within the pages. If a page is different enough, it is backed up and rolled back in the event that a ransomware attack plays out. However, it does not cover situations outside of ransomware, and its history is extremely short, only storing a single backup per page. RansomBlocker [30] is a separate system which detects malware by looking at write patterns and sends signals to other recovery tools which can reinstate the hard drive. As it lacks its own recovery option, it is reliant on systems like ours.

Flash-based data recovery from malware attacks. Data restoration from the general malware could be more challenging compared to the ransomware. This is due to the heterogeneous nature of the malware. Bolt [21] enabled the system restoration after the bare-metal malware analysis in a strictly controlled environment. System restoration turns easier as Bolt clearly knows when the malware starts to function. MobiDOM [31] was developed to address malware threats in everyday scenarios where the defender lacks prior knowledge of when the malware might appear. A key design of MobiDOM is incorporating a secure malware detector which can function correctly even if the the OS is compromised, by utilizing the isolated nature of trusted execution environment and the flash translation layer. The malware detector will securely collaborate with the FTL to enable data restoration after the malware is detected. MobiDR [32] can ensure accurate restoration of the data at the point of time right before the malware starts to corrupt the data, by integrating both the FTL-based recovery and the remote version control system. Huang, et al. [33] uses a similar system to ours, leveraging out-of-place updates to roll back data, but lacks any discussion of preserving pages. In addition, it brings no discussion of detection to the table, and has a higher impact on storage volumes due to its lack of specificity regarding data. This leaves a system which has much worse performance overhead and a much shorter cache life for rolling back data. Ref. [34] Makes use of out-of-place updates in order to make backing up an SSD more efficient, allowing for faster recovery from malware. This system is a collaborative effort between the OS, the FTL, and a Cloud service, all of which work towards database system recovery in an efficient manner.

Flash-based data recovery in critical systems. The existing data recovery designs based on flash memory storage were mostly for computing devices used in non-critical systems. System restoration in critical systems such as cyber-physical systems turned to be even more challenging due to additional considerations such as real-time and safety requirements. Dafoe et al. [35] designed a new framework which can restore the ECU (electronic control unit) firmware in connected and autonomous vehicles. In their design, the repair coordinator running in the secure world of Arm TrustZone was notified by a detector ECU that the ECU it resided had been compromised. This notification was sent stealthily to the TrustZone via the CAN bus. The repair coordinator then worked with the local FTL to restore the mapping data, which are small in size and can be restored efficiently in real time. Note that all the communications remain transparent to the compromised OS by leveraging steganography. After recovering the mapping data, the ECU firmware is restored to its healthy state prior to the compromise. ref. [36] describes a system which can recover from power failures (and other associated catastrophic failures of an SSD) by leveraging the out-of-bounds area described in Section 2.5 in order to store extra metadata. This increases the reliability of the OS and also allows the flash to use error-correcting codes to work around complete failures. However, it assumes that these are not malware related and rolls back data indiscriminately.

ACRecovery is different from all the aforementioned data recovery designs as it mainly concerns on restoring the access control related metadata, which are hidden among the giant raw data in the flash memory and require careful disk forensic analysis in order to be extracted. Mostly importantly, as the OS is compromised, ACRecovery needs to purely rely on analyzing the raw flash memory data to distill the access control permission data created by the OS, which brings extra technical challenges. In addition, ACRecovery uses the specificity of its target to its advantage, allowing for smaller memory overhead than other systems designed for this type of use.

4. Models and Assumptions

System model. We consider a computing device using flash memory as external storage. This includes servers and personal computers which use solid-state drives (SSD), and mobile/IoT devices which use SD/miniSD/microSD cards, MMC/eMMC cards, or UFS cards.

Adversarial model. The adversary, e.g., a piece of malware, is able to compromise the OS of the victim device, obtaining root access. Abusing this high system privilege, the adversary is motivated to arbitrarily modify the access control for files across this device. For example, it may create a hidden user and grant this user arbitrary permissions of files, or it can delete or modify the existing file permissions of legitimate users. The device owner will be aware of this misbehavior over time (e.g., utilizing a low-level malware detection solution [31]). After having removing the malware, the user then needs to recover changes made by the adversary to the access control metadata. This seems impossible conventionally, because the adversary has compromised the OS before and tried to eliminate any traces accessible to the OS which may be relied on to get the access control metadata restored.

Assumptions. Our design is based on a few assumptions: (1) The structure of the file system is intact. This assumption is reasonable as it would require the adversary with very high levels of knowledge about the operating system to compromise the file system's structure without completely destroying the victim device. (2) The flash translation layer (FTL) is secure. The FTL stays between the OS and the flash memory chips and it is typically isolated by the storage hardware from the OS. This hardware-level isolation ensures the security of the FTL. Re-flashing the flash memory firmware may disrupt this assumption, which however is destructive in nature and would result in a wipe of the drive.

5. ACRecovery

5.1. Main Design of ACRecovery

The design of ACRecovery is formed of three major components: setup, monitoring, and recovery. Setup aims to create a system by which we can track writes across the drive. Monitoring checks whether a page contains access control metadata when a write occurs and caches it if so. Lastly, recovery rolls the cached pages back, but excludes non access control related data. Given that these three aspects all require data directly from the drive, each filesystem also has its own challenges due to the way that it stores its metadata. This means that we must perform forensics to determine the filesystem and various aspects thereof. Note that this can be applied to other aspects of the filesystem and pieces of metadata as well, but we specifically focus on the access control related metadata.

5.1.1. Setup

The setup phase aims to establish a method for us to check whether a write is going to affect access control data. In order to do this, we must scan both the Master Boot Record (MBR) of the drive as well as the filesystem itself. We open with a scan of byte 0x1C2 of the drive, which contains a code representing which filesystem is stored on the first partition slot of the drive. For example, code 0x83 represents EXT, while a series of codes represent versions of FAT and NTFS. We then check bytes 0x1C5-0x1C9 of the drive. This contains the LBA address of the starting block of the partition [13]. These two pieces of data combine to tell us what kind of partition to scan and where it begins on the drive, allowing more filesystem-dependent portions of setup to run. From there, we implement a system which catalogues the pages on which file metadata can be stored. Once this data is secured, it can be monitored. In addition, this data does not need to be copied on-drive, as the method for finding it is consistent and can be repeated even after a reboot. This means that setup should have an overall storage overhead of zero, while taking up a minimal amount of memory.

5.1.2. Monitoring

Most monitoring systems follow the pattern of Algorithm 1. When a write is triggered, the isWritingPermission function gets called. This function is filesystem dependent due to the variety of sizes and shapes that access control data takes. In general, the function is only applied on a page by page level in monitoring due to the fact that writes are not considered expensive and must be efficient. Even so, the number of pages which contain access control data varies from filesystem to filesystem. For instance, in EXT, access control

data is stored in a series of groups across the drive, while NTFS stores all of its data in two to three places. Ultimately, isWritingPermission is treated as a blackbox which takes in the filesystem type, the address being written, and the data being written to it, and returns true if and only if the write affects access control data. In the event that it does, we also must prevent the page from being marked for deletion. This can be performed in a variety of ways, such as adding it to a cache, marking group headers in the OOB, etc. Thus, when the page is written, the previous address is accessible on subsequent reads. After the write is performed, the user is able to roll back the writes which were tracked at any point desired by sending a signal to the drive.

Algorithm 1 FTL_Write(pageaddress addr, byte data) /* Write data to logical page addr */

```
1: int block, page //Will hold the block and page ids for addr
2: PMT_Search(addr, &block, &page)
3: if isWritingPermission(addr,data,filesystem_type) && !isRollingBack then
4:     CACHE_Add(currentWrites, addr) //Add the address to our cache of permissions writes
5:     data.OOB.append(PHYSICAL_ADDRESS)
6: end if
7: drive.write(block, page, data)
8: return
```

5.1.3. Rollback

At a certain point, the user may want to undo recent changes to access control data. This generally follows Algorithm 2. The isPermissionByte function takes in the byte desired, the whole page being questioned, and the filesystem type, and returns true if the byte in question is related to access control data. Noting that the isPermissionByte function is highly dependent on the filesystem type, and detailed discussion can be found in Section 5.2. As the loops run, the permissions data in the new page is replaced with that of the old page without editing any other data. Following this, the page is written to its original location, effectively replacing any newly written permissions data with the oldest version present in the cache. Once each page in the cache has been written, it is cleared to prepare for new writes. After the rollback has run, the access control data should be rolled back to its state at the beginning of the cache, with all other data preserved. If desired, code could easily be written to allow the user to select individual writes to roll back, allowing for a more powerful tool. An example of this is shown in Figure 4. Malicious ACL data is written to page 12, causing its physical address to shift to page 73. Because monitoring detects this, the user can then issue a rollback command. This causes the access control data data from page 73 to be overwritten with the ACL data from page 72, and the final result is written to a new page 77.

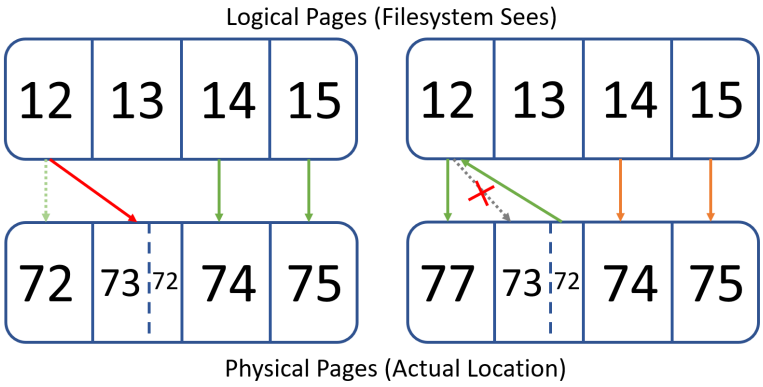

Figure 4. Example of rolling back changes to a page.

Algorithm 2 Rollback() /* roll back all recent access control changes */

```
1: DATA* data, oob* OOB // Used for reading in the page data (data) and out-of-bounds
   area (OOB).
2: isRollingBack ← TRUE
3: for page in cache do
4:     FTL_Read(page, &data, &OOB)
5:     FTL_Read(OOB[2], &data2, &OOB)
6:     for byte in page do
7:         if isPermissionByte(byte, data, filesystem_type) && data[byte] != data2[byte]
   then
8:             data[byte] ← data2[byte]
9:         end if
10:    end for
11:    FTL_Write(page, data2)
12: end for
13: CLEAR_CACHE(pages)
14: Return
```

5.2. Implementing ACRecovery *in EXT Filesystems*

In this section, we discuss the implementation of ACRecovery when the EXT filesystems are deployed on the device. We will discuss filesystem specific aspects and challenges, then tie it back to the general design presented previously. The description applies to both EXT2 and EXT3, allowing for a single unified codebase for easier user access. The journaling aspect of EXT3 allows for more write security, but does not interfere with the way that metadata is stored. This allows for the code to stay the same.

5.2.1. EXT: Challenges

Since EXT stores file data in groups of inodes across the drive, the primary issues with EXT are going to involve location and tracking of the pages responsible for holding inodes. There are a series of challenges that face us when working on EXT. We use EXT2 for its simplicity, but adjustments for more modern systems should be minor. As such, challenges for EXT are:

1. How big is a group of inodes? Since inode groups are where access control data is stored, we need this number. We need it specifically in pages, as we need to accurately track inode groups without storing each and every page which contains an inode in cache, which would use a large amount of memory. This will help us saving said memory when searching for whether a page contains inodes during the monitoring phase.
2. How many groups of inodes are there in this filesystem? We need to know how large of a cache we need, and multiplying the size of a group by the number of groups yields the result. When the cache is initialized, we want to have an exact size in mind since internal memory usage on a drive of this nature should be kept to a minimum. Normally, this number is relatively low (<100), which results in low memory usage accordingly if handled well. The search given enables us to size the cache exactly as desired.
3. Where is each group of inodes actually stored on the drive? To track the access control data, one must know where it is located. Groups are stored at various places across the drive, and each one needs to have its descriptor checked in order to gather its 'inodes' locations. These descriptors are stored immediately after the superblock as described in Section 2.2, and can be found by looking relatively early on in the descriptor. Notably, we need to store these locations in ascending order, as this will enable us to binary search the cache in the monitoring phase. Thankfully, the block groups are stored in ascending order, so we do not need to perform any kind of sorting following the setup phase for our desired order to be maintained.

5.2.2. EXT: Setup

Figure 3 outlines the procedure for EXT setup. This primarily aims to establish the location and size of each inode group in the system. We open by reading the "superblock", which stores a large amount of details regarding portions of the drive. We are primarily interested in the following pieces of of data in the superblock:

1. The size of a "block" (not related to the term used in Section 2.5), stored at byte 24 with length 4 bytes. This will only be used to turn data gathered in the superblock into data which is usable by the FTL.
2. The number of inodes in an inode group, stored at byte 40 with length 4 bytes. This is used in calculations for the size of an inode group.
3. The total number of inodes in the filesystem, stored at byte 0 with length 4 bytes. This is used to calculate the number of inode groups, which we need in order to properly initialize the cache.

We then check the 4-byte word at offset 76, as it contains the major version. If this is greater than 1, we need to check the 2-byte word at offset 88 to get the size of an inode. Otherwise, the size is set strictly at 128 [9]. Often, inode size is set to 256 for more modern systems.

These pieces of information are combined to calculate the following:

1. The number of inodes in a page, given as page_size/inode_size. Used in calculations for the number of pages an inode group takes.
2. The number of inode groups, given as total_num_inodes/inodes_per_group. This helps us initialize the cache.

Using the gathered pieces of information and the previously known constant of page size, we can assemble a picture of how large an individual block group's worth of inodes is by dividing the number of inodes per group by the number of inodes per page. From there, we read each of the individual block group descriptors. These are located immediately after the superblock, and the start location of their inodes is precisely 8 bytes from the start of the descriptor. Since descriptors are exactly 32 bytes long [9], we can easily find all of the start locations with the information previously provided. All of these combine to create a system which caches the locations of the start of each group of inodes and their size, allowing monitoring to take over.

5.2.3. EXT: Monitoring

EXT's monitoring system is relatively simple, as the setup phase has done most of the heavy lifting. However, it is worth noting the particulars. Given that we have cached the size of an inode group and the starting location of each, we can run a binary search over the cache when a write is ordered. We return true if the target page is within the chosen size of one of the cached values, and false otherwise. If false is returned, the operation continues with no modifications. Following a true result, the target is added to the write cache, the page is marked as in use, and the old location of the page is added to the OOB. From this point, the write continues as normal. Given that checking for non-permission changes would require a thorough scan of two pages' worth of data, this is left to the recovery phase, and monitoring ends here.

Algorithm 3 Setup(initialPage) /* Read EXT data from a partition starting at initialPage */

```
data_buffer ← FTL_Read(initialPage) //Read the start of the partition and place the data in a variable
unsigned int blockSize ← data_buffer.getWord(1048) //Size of a block in the EXT system
unsigned int inodesPerGroup ← data_buffer.getWord(1064) //# of inodes in an inode group
unsigned int totalInodes ← data_buffer.getWord(1024) //Total number of inodes
if data_buffer.getWord(1100) >0 then
    unsigned short inodeSize ← short(data_buffer.getWord(1112)) //Size of an inode, in bytes
else
    unsigned short inodeSize ← 128
end if
unsigned int inodesPerPage = int(ceiling(double(PAGE_SIZE/inodeSize))) //# of inodes in a flash page
unsigned int groupCount ← int(ceiling(double(totalInodes/inodesPerGroup))) //# of groups on the drive
unsigned int bgdStart ← max((1024 << blockSize) * 2, 2048) //Block group descriptors start
unsigned int currentOffset ← bdgStart%PAGE_SIZE //Used to track the starting byte of our current block group descriptor
unsigned int newPage ← initialPage //Page we're checking
for int i ← 0; i < groupCount; i++ do
    unsigned int currentStart ← data_buffer.getWord(currentOffset + 8) //Start of the currently checked block group's inodes.
    unsigned int startPage = initialPage + int(currentStart * ((1024 << blockSize)/double(PAGE_SIZE))) //Convert the result to a flash page ID
end for
permissionCache.add(startPage)
currentOffset += 32
if currentOffset >= PAGE_SIZE then
    currentPage++
    data_buffer ← FTL_Read(currentPage)
    currentOffset %= PAGE_SIZE
end if
Return
```

5.2.4. EXT: Rollback

Rollback follows the pattern shown in Figure 4. Permissions data is the first 3 bytes of an inode. Since we harvested the length of an inode in the setup portion, we are able to ignore bytes which are not in the first three bytes of the inode. However, we must perform one further check: If the type setting of the file (the first 4 bits of the first byte) are out of the range 0x10-0xCF, then we know that the file was either created or destroyed by the previous operation [9]. This value being invalid indicates that the file was either created (if buf2 is invalid and buf1 is not) or deleted (if buf1 is invalid and buf2 is not) by the write being rolled back. Once a page has been fully vetted, we may write the newly created page to its original location by using the address we cached for writing. This is a simple, yet expensive, operation and as such is left to rollback. In addition, each page must be written, which takes up further resources. This algorithm is presented as an example of why recovery is presented as expensive and why detecting whether a change directly affects permissions data is left up to an expensive function.

Algorithm 4 Rollback() /* Take our cache of pages and revert access control changes */

```
PAGE page, page2 //Used to hold page data
OOB spare //Used to hold OOB data
for int i ← 0; i < permissionCache.size; i++ do
    data_buffer, spare ← FTL_Read(permissionCache.contents[i])
    page2 ← spare[2]
    buffer2, spare ← FTL_Read(page2)
    for int j ← 0; j < PAGE_SIZE; j+ = INODE_SIZE do
        if data_buffer[j + 1]! = buffer2[j + 1] then
            data_buffer[j] ← buffer2[j]
            data_buffer[j + 1] ← buffer2[j + 1]
        end if
    end for
    FTL_Write(permissionCache.contents[i], data_buffer)
end for
Return
```

6. Experimental Evaluation

6.1. Experimental Setup

We chose to implement our design by modifying an existing open-source FTL software known as OpenNFM [37]. OpenNFM contains full C code for an FTL implementation, allowing for modification to any relevant read/write functions and further additions to the project. Our experiments were performed using an LPC-H3131 board (512 MB storage, 180 MHz controller with 192 kB RAM) [38] on a virtual machine running Kali Linux. The VM was allocated 2 cores of an intel i9-10900KF (5.0 GHz overclock) [39] and 4 GB of 2.1 GHz RAM. The drive was flashed by using the board's reset button and downloading a compiled binary to it [40]. We could then install the relevant filesystem on top of OpenNFM. We elected to perform two tests:

1. Throughput test, particularly in write speed. Since monitoring directly affects the write functionality of the drive, a test of throughput is needed in order to check whether our system dramatically impacts usability.
2. Time to rollback. With our system, it is good to know how long a queue of a given size takes to revert to its original state. This will give us a sense of how major a change the rollback operation gives.

These two tests should be able to give an indication of the usability of the system and its impact on the FTL as a whole. If the results of our experiments are positive, the design can be quickly adapted to a publishing of the system. On the other hand, negative results may indicate that the design needs further adjustment before being given to the general public.

6.2. Experimental Results

Throughput. The results of the throughput tests are displayed in Tables 1 and 2. Results show a slight decrease in write speeds (and an extremely slight increase in read speeds in some cases, but this is likely caused by system usage at the time the test was run or any number of other factors). This indicates a small, but not problematic loss of performance compared to an unmodified instance, or it could be an indication that the impact of modifications on the drive is irrelevant compared to other unseen factors like system use. In either case, a system like ours is shown to be fully feasible for current implementation.

Rollback time. Rollback time results are shown in Figures 5 and 6. The rollback times shown follow a mostly linear pattern, with a slight drop-off from the line at a cache size of 200. This is the expected pattern given that all algorithms provided are in linear time. In addition, rollback proves to be a short process, with a queue of 200 taking less than 2 s on a consistent basis. As the system is convenient to use in addition to not taking a large toll on performance, this once again indicates that the system can be implemented with no major changes.

Table 1. Throughput comparisons (EXT2).

Type	Unmodified	Modified
Seq-Read	1450 kB/s	1454 kB/s
Rand-Read	1022 kB/s	1029 kB/s
Seq-Write	912 kB/s	880 kB/s
Rand-Write	788 kB/s	773 kB/s

Table 2. Throughout comparisons (EXT3).

Type	Unmodified	Modified
Seq-Read	1464 kB/s	1456 kB/s
Rand-Read	1054 kB/s	1052 kB/s
Seq-Write	1291 kB/s	1218 kB/s
Rand-Write	845 kB/s	827 kB/s

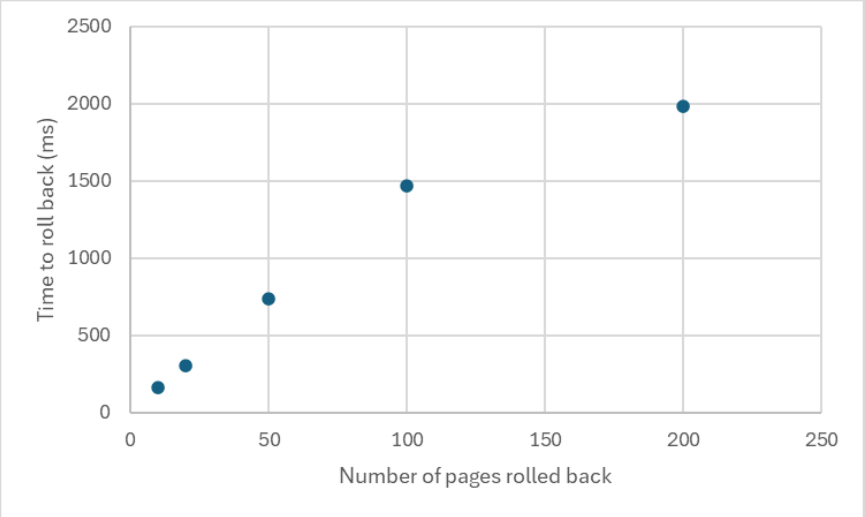

Figure 5. The time needed for rollback (EXT2).

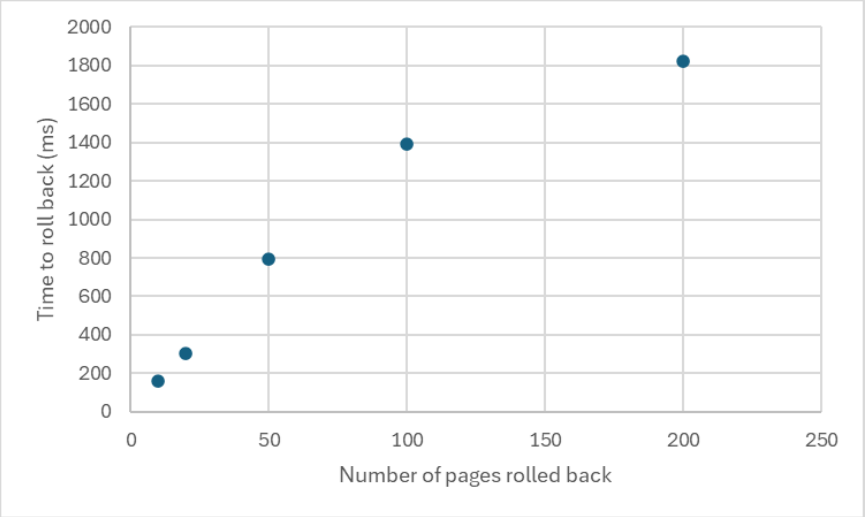

Figure 6. The time needed for rollback (EXT3).

7. Discussion

Naturally, our implementation does not cover every possibility and permutation. There are several future directions and improvements that can be made to the system. We

cover several such improvements and challenges here in detail, and anticipate that even more can be made in the future to improve on the work.

Danger and abuse of power. Naturally, a system with the power to undo user actions should be heavily restricted due to its potentially compromising nature. We propose use of a manual hardware switch which can trigger rollback, as the only way to completely guarantee that a command cannot be executed without physical access to the machine is to hard-wire it in. Of course, this comes with its own set of issues. Bringing a physical device relies on the user not forgetting to do so, and needing to retrieve an object increases the time between a rollback being requested and actually carried out. However, since the actions this program is capable of are so powerful, it is necessary to take adequate precautions.

OS write caches. One issue that came up during testing is the fact that Linux commands often cache writes rather than committing them immediately upon instruction [41]. For example, if a file is copied and then permissions are immediately changed, the original version with unchanged permissions is never written to the drive and thus never cached. Notably, re-mounting the drive commits all pending writes, allowing us to work around this problem in testing. This feature has both upsides and downsides for our work. Favorably, this means that repeating a singular command to change the permissions data will likely only commit one change to the drive, thus preventing one potential strategy to clear the cache. However, as mentioned, an attack that targets a file immediately after creation may not be reversible due to file creation.

Partitions. The algorithm given in Section 5.1 for locating the partition on the drive is configured to find the first partition. However, future partitions can also be found by changing the following addresses:

1. System type: 0x1c2 - > 0x1d2 (2nd partition), 1e2 (3rd partition), or 1f2 (4th partition) [13]
2. LBA start: 0x1c6 - > 0x1d6 (2nd partition), 1e6 (3rd partition), or 1f6 (4th partition) [13]

This allows users to select which partition on the drive is protected in case of situations such as dual booting for multiple OSes.

NTFS. New Technology File System [42] was designed for use in Windows NT and is currently used as its primary filesystem after phasing out FAT in the 1990s. NTFS is stored as a series of files on the drive, with even important data such as boot records and metadata stored in the same format. The latest version is NTFS 3.1, released in 2001. This system features a boot record, followed by a list of files and their data. This list is known as the Master File Table, or $MFT, and is stored as the second file on the drive. The file preceding the MFT, $Boot, contains diagnostic information for the drive. NTFS uses a system of "clusters" of "sectors" of bytes, which are used to catalogue files. It is worth noting that these clusters are usually larger than one page of flash storage. Each file is given a certain number of clusters which it has control of and manages. Access control data is stored in two categories: A "read-only" and "hidden" bit, stored in two places across a file's record, and the security entry, stored in a file called $Secure. The former category means that a system which handles NTFS must track the data in $MFT, while the latter means we must track the contents of $Secure. This is doable, as this data is stored in contiguous sections, but some aspects of NTFS are undocumented due to its nature of being designed for Windows. As such, systems which handle NTFS would need some expertise from Windows programmers.

FAT. File Allocation Table based filesystems (FAT) [43,44] are the third most popular filesystem for drive use, mostly utilized by older Windows computers. They are formatted as a boot record, followed by a table listing the clusters on the drive and which files occupy them. Directories are stored as a special type of file on the drive, specifically a table of entries which describe files and metadata, and are stored in the standard data section of the drive. This data section begins with a root directory entry, which contains pointers to both its files and any subdirectories. These are 32-byte entries which contain both the location of the actual relevant data and any contained metadata, such as read-only nature, creation time, etc. These subdirectories each have their own file and subdirectory entries,

leaving the directory structure as a tree. When a new directory is to be loaded, the tree is traversed using a path provided. Any metadata related to a file is stored with that file's directory entry, which means that for a particular file to be checked, it must be checked in the tree [45]. We now provide a summary of FAT's challenges and how the phases for it might be implemented:

Page saving. There are a variety of options which can be used to keep pages from being erased, and we need to utilize one in order to prevent data we are tracking from being removed. Dafoe, et al. [35] keeps a cache of pages which are currently in use by the system, preventing them from being erased at all during the time where pages are stored. Our test design used a system where the number of dirty pages in a block is only incremented when permissions are not being written, which dramatically decreases the odds that the old blocks are erased. Our application uses a differing method which attempts to use less memory. We instead insert a small boolean check into the PMT_Update function which does not increment the unused block counter for early pages, which should lower the chance that an old page is selected. This saves time and memory, but comes with a potential risk of increased wear leveling inequality.

Boot proofing. Given that we need to use a cache to track pages which have been changed, there needs to be a method of boot-proofing the system so that a shutdown of the drive does not cause us to lose progress. Dafoe, et al. [35] uses an extra page on the flash device to write down the cache in the event that the drive is shut down, an option that we can take as well in order to keep our data safe. We would need to store both the current cache of pages which have been written and any data needed for overwrite prevention. This page can be read to recover the state of the tracking at any given point. We could also add the setup phase data, but it is not strictly necessary as all of the setup phase is doable at any point after the partition is created. In addition, if boot-proofing can be made more efficient, it is hypothetically possible to design a system that fully recovers the entire filesystem from complete corruption. This is possible owing to the fact that we are already recovering filesystem data, and could simply remove some filters during rollback and expanding the range which the cache searches. EXT in particular is susceptible to this, as inodes make up the majority of the filesystem's data.

GPT Partition handling. Some modern drive setups will use a system called the GUID partition table (GPT) rather than the MBR for drive partitions. This system has an initial MBR entry whose partition type is 0xEE, allowing for ACRecovery to adjust accordingly. The GPT system does not have an OS Type header, but does still contain the location of the partition. As such, a GPT-compatible ACRecovery implementation would need to check the partitions for magic numbers to determine OS type rather than using the MBR entries. This is doable, however our test machines did not use GPT and such a system was not implemented.

EXT4. EXT4 is the newest version of the EXT filesystem. This added several features, including an extended superblock, dynamic groups, and more. One feature added to EXT4 is metadata checksums for all applicable inodes. This uses a crc32c algorithm to ensure the integrity of metadata across the filesystem. While good for integrity, this makes our system much more difficult to implement, especially when taking write caching (Section 7) into account [11]. Since crc32c is seeded, we are unable to generate new checksums for the system. In the event that a write changes both the permissions of a file and some other piece of its metadata, a system like ours must either undo all of the applicable changes or none of them due to the checksum. As such, any implementation of our system for EXT4 must be limited, though some semblance of the system can still be implemented. During rollback, a page would need to be checked for significant changes outside of permissions (minor changes like the last time modified can be safely reverted without major issue). If no significant changes are detected, the page can then be rolled back. However, if other changes (for instance, the size of the file) are made, the user may need to be informed and perform the changes manually.

Detection in the FTL. Given that the OS is compromised, it may be impossible for the user to deduce that their system is infected at all. Since we assume that the malware is removed in our system, it is worth acknowledging possible detection methods in the FTL. Chen et al. [31], as well as several other works [24,25,28], do this for mobile devices by scanning for noticeable access patterns across the FTL which lead to malware attacks. It is theoretically possible to transfer a system such as this into the FTL from the trusted app layer by downscaling the application to fit the limited processor space available. More work needs to be done in this area, but it should still be well within possibility to perform this detection automatically in the future without trusting the OS at all. This would enable users to be secure without even having to think about it in most scenarios.

Future work. Several current weaknesses in the system need to be adjusted for future iterations of the work. These include:

1. Boot-proofing. As discussed in the previous section, we must boot proof the system. Once this has been done, several attacks on our work become invalid as we can simply recover the state of the system after a reboot. Until then, a reboot clears the cache, possibly preventing some changes from being rolled back.
2. Byte-specific monitoring. While our current system leaves byte-specific checks for the rollback phase, it is hypothetically possible to do so during monitoring. This would save a large amount of cache space as false positives become impossible. However, a method for doing so efficiently has yet to be found.
3. Automatic Setup. Right now, the setup operation is performed by a command sent from the user. However, it may be possible for the FTL to detect a specific write pattern which corresponds to the completion of a partition on the drive. More work will need to be done in this area to confirm or deny this possibility.

8. Conclusions

In this work, we have designed ACRecovery, a novel scheme which can restore the access control system compromised by a high-privilege adversary. This is accomplished entirely within the flash translation layer, without any assistance from the operating system. By performing forensic analysis over the raw data in the flash memory, we can extract the original access control metadata of the OS, enabling the rollback of the most recent changes on them (due to the out-of-place update mechanism in the FTL). Such a design is necessary for defending against the OS-level adversary which can compromise the operating system and arbitrarily modify the access control metadata in the OS. We have presented the implementation of ACRecovery on EXT2 and EXT3 filesystems, and perform experimental evaluations which show that ACRecovery can roll back the compromised access control metadata in the OS efficiently, without incurring a significant impact on the lifespan and throughput of the flash storage drive. This pioneering research paves the way for future research on filesystem forensics within the FTL, thereby benefiting the data recovery community. Potential followup works include expanding the system to support the FAT and NTFS filesystems as well as extra optimizations of the current design.

Author Contributions: Conceptualization, C.R. and B.C.; methodology, C.R. and B.C.; validation, C.R. and B.C.; formal analysis, C.R. and B.C.; investigation, C.R. and B.C.; writing—original draft preparation, C.R. and B.C.; writing—review and editing, C.R. and B.C.; project administration, B.C.; funding acquisition, B.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by National Science Foundation under grant number 2225424-CNS, 1928349-CNS, and 2043022-DGE.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Our developed prototype and demo video are publicly available in https://github.com/NotoriousADC/flashback (accessed on August 5th, 2024).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ACL	Access Control List
CAN	Controller Area Network
ECU	Electronic Control Unit
EXT	EXTended file system
FAT	File Access Table
FTL	Flash Translation Layer
GPT	GUID Partition Table
NTFS	New Technology Filesystem

References

1. Foundation, T.L. Understanding Linux Permissions. Available online: https://www.linuxfoundation.org/blog/blog/classic-sysadmin-understanding-linux-file-permissions (accessed on 22nd May 2024).).
2. Critical RCE Vulnerability in Linux Kernel Let Hackers Compromise the Entire Systems Remotely. Available online: https://cybersecuritynews.com/linux-kernel-bug-3/ (accessed on 6th May 2024).
3. Microsoft. CVE-2021-34535. Available online: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2021-34535 (accessed on 21st May 2024).
4. NIST. CVE-2018-4121. Available online: https://nvd.nist.gov/vuln/detail/CVE-2018-4121 (accessed on 23rd May 2024).
5. Man Chmod. Available online: https://linux.die.net/man/1/chmod (accessed on 21st May 2024).
6. Gupta, A.; Kim, Y.; Urgaonkar, B. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the ACM*; ACM: New York, NY, USA, 2009; Volume 44.
7. Hardock, S.; Petrov, I.; Gottstein, R.; Buchmann, A. From In-Place Updates to In-Place Appends: Revisiting Out-of-Place Updates on Flash. In Proceedings of the 2017 ACM International Conference on Management of Data, New York, NY, USA, 14–19 May 2017; SIGMOD '17; pp. 1571–1586. https://doi.org/10.1145/3035918.3035958.
8. Wirzenius, L.; Oja, J.; Stafford, S.; Weeks, A. Filesystems. Available online: https://tldp.org/LDP/sag/html/filesystems.html (accessed on 22nd May 2024).
9. OSDev Wiki. 2022. Available online: https://wiki.osdev.org/Ext2 (accessed on 21st May 2024).
10. ext3 Filesystem. 2024. Available online: https://docs.kernel.org/filesystems/ext3.html (accessed on 21st May 2024).
11. ext4 Filesystem. 2024. Available online: https://www.kernel.org/doc/html/v4.19/filesystems/ext4/index.html (accessed on 21st May 2024).
12. Arpaci-Dusseau, R.H.; Arpaci-Dusseau, A.C. *Operating Systems: Three Easy Pieces*; Arpaci-Dusseau Books, Madison, WI, USA : 2023; Chapter 3.
13. OSDev Wiki. 2023. Available online: https://wiki.osdev.org/MBR (accessed on 20th May 2024).
14. Gruenbacher, A. POSIX Access Control Lists on Linux. In Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC 03), San Antonio, TX, USA, 9–14 June 2003.
15. Govindavajhala, S.; Appel, A.W. *Windows Access Control Demystified*; Princeton University: Princeton, NJ, USA, 2006.
16. Liu, Z. SSD and HDD Statistics from EaseUS. 2023. Available online: https://www.tomshardware.com/news/ssd-and-hdd-statistics-from-easeus (accessed on 22nd May 2024).
17. Cai, Y.; Ghose, S.; Haratsch, E.F.; Luo, Y.; Mutlu, O. Error Characterization, Mitigation, and Recovery in Flash-Memory-Based Solid-State Drives. *Proc. IEEE* **2017**, *105*, 1666–1704. https://doi.org/10.1109/JPROC.2017.2713127.
18. Luo, Y.; Lin, M. Flash translation layer: A review and bibliometric analysis. *Int. J. Intell. Comput. Cybern.* **2021**, *14*, 480–508.
19. Kim, J.; Kim, J.M.; Noh, S.H.; Min, S.L.; Cho, Y. A space-efficient flash translation layer for compactflash systems. *IEEE Trans. Consum. Electron.* **2002**, *48*, 366–375.
20. Chung, T.S.; Park, D.J.; Park, S.; Lee, D.H.; Lee, S.W.; Song, H.J. A survey of Flash Translation Layer. *J. Syst. Archit.* **2009**, *55*, 332–343. https://doi.org/10.1016/j.sysarc.2009.03.005.
21. Guan, L.; Jia, S.; Chen, B.; Zhang, F.; Luo, B.; Lin, J.; Liu, P.; Xing, X.; Xia, L. Supporting Transparent Snapshot for Bare-metal Malware Analysis on Mobile Devices. In Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, 4–8 December 2017.
22. Wei, M.Y.C.; Grupp, L.M.; Spada, F.E.; Swanson, S. Reliably erasing data from flash-based solid state drives. In Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11), San Jose, CA, USA, 15–17 February 2011; Volume 11.
23. Huang, J.; Xu, J.; Xing, X.; Liu, P.; Qureshi, M.K. FlashGuard: Leveraging Intrinsic Flash Properties to Defend Against Encryption Ransomware. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; ACM: New York, NY, USA, 2017; pp. 2231–2244.

24. Wang, P.; Jia, S.; Chen, B.; Xia, L.; Liu, P. Mimosaftl: Adding secure and practical ransomware defense strategy to flash translation layer. In Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy, Richardson, TX, USA, 25–27 March 2019.
25. Baek, S.; Jung, Y.; Mohaisen, A.; Lee, S.; Nyang, D. Ssd-insider: Internal defense of solid-state drive against ransomware with perfect data recovery. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2–6 July 2018.
26. Baek, S.; Jung, Y.; Mohaisen, D.; Lee, S.; Nyang, D. SSD-assisted ransomware detection and data recovery techniques. *IEEE Trans. Comput.* **2020**, *70*, 1762–1776.
27. Chen, N.; Dafoe, J.; Chen, B. Poster: Data Recovery from Ransomware Attacks via File System Forensics and Flash Translation Layer Data Extraction. In Proceedings of the 2022 ACM Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022.
28. Min, D.; Park, D.; Ahn, J.; Walker, R.; Lee, J.; Park, S.; Kim, Y. Amoeba: An autonomous backup and recovery ssd for ransomware attack defense. *IEEE Comput. Archit. Lett.* **2018**, *17*, 245–248.
29. Min, D.; Ko, Y.; Walker, R.; Lee, J.; Kim, Y. A content-based ransomware detection and backup solid-state drive for ransomware defense. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *41*, 2038–2051.
30. Park, J.; Jung, Y.; Won, J.; Kang, M.; Lee, S.; Kim, J. RansomBlocker: A low-overhead ransomware-proof SSD. In Proceedings of the 56th Annual Design Automation Conference 2019, Las Vegas, NV, USA, 2–6 June 2019; pp. 1–6.
31. Chen, N.; Chen, B. Defending against OS-Level Malware in Mobile Devices via Real-Time Malware Detection and Storage Restoration. *J. Cybersecur. Priv.* **2022**, *2*, 311–318.
32. Xie, W.; Chen, N.; Chen, B. Enabling Accurate Data Recovery for Mobile Devices against Malware Attacks. In Proceedings of the 18th EAI International Conference on Security and Privacy in Communication Networks, Virtual Event, 17–19 October 2022.
33. Huang, P.; Zhou, K.; Wu, C. ShiftFlash: Make flash-based storage more resilient and robust. *Perform. Eval.* **2011**, *68*, 1193–1206. Special Issue: Performance 2011. https://doi.org/10.1016/j.peva.2011.07.010.
34. Son, Y.; Choi, J.; Jeon, J.; Min, C.; Kim, S.; Yeom, H.Y.; Han, H. SSD-Assisted Backup and Recovery for Database Systems. In Proceedings of the 2017 IEEE 33rd International Conference on Data Engineering (ICDE), San Diego, CA, USA, 19–22 April 2017; pp. 285–296. https://doi.org/10.1109/ICDE.2017.88.
35. Dafoe, J.; Singh, H.; Chen, N.; Chen, B. Enabling Real-Time Restoration of Compromised ECU Firmware in Connected and Autonomous Vehicles. In Proceedings of the 2023 EAI International Conference on Security and Privacy in Cyber Physical Systems and Smart Vehicles, Chicago, IL, USA, 12–13 October 2023.
36. Jung, S.; Song, Y.H. Data loss recovery for power failure in flash memory storage systems. *J. Syst. Archit.* **2015**, *61*, 12–27. https://doi.org/10.1016/j.sysarc.2014.11.002.
37. Code, G. OpenNFM. 2011. Available online: https://github.com/IMCG/opennfm (accessed on 30th September 2024).
38. Olimex LPC-H3131. Available online: https://www.olimex.com/Products/ARM/NXP/LPC-H3131/ (accessed on 20th May 2024).
39. Intel Core I9-10900KF. Available online: https://ark.intel.com/content/www/us/en/ark/products/199331/intel-core-i9-10900kf-processor-20m-cache-up-to-5-30-ghz.html (accessed on 20th May 2024).
40. Tankasala, D.; Chen, N.; Chen, B. Creating A Testbed for Flash Memory Research via LPC-H3131 and OpenNFM—Linux Version. 2022. Available online: https://snp.cs.mtu.edu/outreach/OpenNFM-LPC-Ubuntu.pdf (accessed on 20th May 2024).
41. Firmianay. Chapter 16: The Page Cache and Page Writeback. 2021. Available online: https://github.com/firmianay/Life-long-Learner/blob/master/linux-kernel-development/chapter-16.md (accessed on 21st May 2024).
42. Microsoft. How NTFS Works. 2024. Available online: https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc781134(v=ws.10) (accessed on 21st May 2024).
43. FAT File Systems. Available online: https://www.ntfs.com/fat_systems.htm (accessed on 21st May 2024).
44. ELMChan. Available online: exFAT filesystem. http://elm-chan.org/docs/exfat_e.html (accessed on 22nd May 2024).
45. OSDev Wiki. 2024. Available online: https://wiki.osdev.org/FAT (accessed on 5th May 2024).