# Is there a "rowhammer" for MLC NAND Flash SSDs? An analysis of filesystem attack vectors

Anil Kurmus Nikolas Ioannou Matthias Neugschwandtner Nikolaos Papandreou
Thomas Parnell
*IBM Research – Zurich*

## Abstract

Rowhammer demonstrated that non-physical hardware-weakness-based attacks can be devastating. In a recent paper, Cai et al. [2] propose that similar attacks can be performed on MLC NAND flash-based SSDs, with potentially devastating effects as well. In this paper, we discuss the requirements for a successful, full-system, local privilege attack on SSDs and show a filesystem based attack vector, which we demonstrate. In particular, to motivate the assumptions of the filesystem-level attack, we show the attack primitive that an attacker can obtain by making use of cell-to-cell interference is quite weak, and therefore requires a carefully crafted attack at the OS layer for successful exploitation.

## 1 Introduction

When developing software, hardware aspects are typically abstracted and security primitives such as resource isolation are taken for granted. In short, modern-day software programming requires little knowledge of the intricacies of the underlying hardware. It took rowhammer [10, 19] to bring back the inner workings of hardware to our attention. Rowhammer exploits a weakness in the deeper layers of memory management to corrupt sensitive memory regions. Recent work has demonstrated how such a non-obvious, complex vulnerability can be exploited to inject faults through Javascript [8], achieve privilege escalation on mobile phones [22], or compromise co-hosted VMs [17].

However, DRAM is not the only place that holds sensitive data that is essential to the correct working of security primitives implemented in software. Persistent storage, organized by means of a file system, grants access to data based on metadata that is stored on disk. In modern-day computers, flash memory has largely replaced spinning disk as the prime persistent storage medium.

Based on a recently published paper by Cai et al. [2] that proposes that rowhammer-like attacks are possible on SSDs but does not present an actual attack, we investigate the feasibility of such attacks on SSDs from the system point of view. In particular, we show that under realistic assumptions on the Flash device behavior and filesystem used, it is possible (albeit challenging) to mount a local privilege escalation attack by leveraging Flash weaknesses. The attack we demonstrate is not "full-system", in the sense that we only demonstrate the filesystem-layer of the attack and assume that a corruption of the underlying Flash media is possible. However, we also describe how the attack could be extended to a full-system attack, which we intend to demonstrate in future work.

More precisely, we use our knowledge of existing reliability mechanisms in SSDs (including ECC), to show that the attack primitive an attacker can obtain from MLC NAND flash weaknesses is a coarse granularity corruption: unlike in rowhammer, where the attacker can flip a single bit, in the case of this attack the attacker can only corrupt one block of data. We then show that this weaker attack primitive (when compared to flipping individual bits, which provides a higher level of control to the attacker) is nevertheless sufficient to mount a local privilege escalation attack.

**Threat model** We assume that the victim system runs a filesystem on top of MLC NAND flash-based SSD. We assume that an attacker has unprivileged, i.e. non-root access to the victim system. This access enables the attacker to cause controlled write accesses to system storage, through the filesystem. A typical scenario would be a login shell for an unprivileged user. We do not assume physical access to the victim system.

**Contributions** The main contributions of this paper are summarized as follows.

- We detail for the first time how a full system, flash-weakness-based, local privilege escalation attack can be mounted.
- We implement and demonstrate the filesystem level part of this attack.

- We discuss possible generalizations and limitations of the attack.

## 2 Background and Related Work

### 2.1 Hardware-based attacks

Table 1 shows a categorization of hardware-based attacks. Foremost, we distinguish between two main categories: physical and non-physical attacks. For a physical attack, direct access to the hardware of a system is required. Examples for such attacks are probing of voltage levels or scraping layers off a silicon chip to reverse-engineer its logic. Less involved examples make use of coolant spray and cold boot to disclose sensitive information from a system's memory banks. On the other hand, non-physical attacks still exploit the hardware of a system, but do not require direct access and are thus potentially more powerful. For example, timing side-channel attacks can be carried out remotely against servers in the cloud.

On an orthogonal level, we distinguish between attacks that compromise confidentiality and integrity of a system. Attacks that compromise confidentiality typically require a read primitive to disclose sensitive information, such as key material in a cache side-channel attack. On the other hand, attacks on system integrity require a write primitive to actively modify and corrupt aspects of a system to change its behavior. A very prominent example is drammer [22], which uses the rowhammer [10, 19] memory write primitive to perform a privilege escalation attack on Android.

The attack shown in this paper falls into the same category of non-physical hardware-based integrity attacks.

### 2.2 Flash weaknesses

MLC NAND Flash exhibits reliability issues related to medium and device characteristics. First, repeated program erase (P/E) cycles on MLC NAND Flash stress the device and gradually deteriorate its reliability [23]. Second, cell-to-cell interference (CCI) is another detrimental effect that takes place during Flash page programming and affects the reliability of NAND devices. The programming voltages applied to the cells of a page that is currently being programmed introduce interference to the cells of the adjacent pages due to capacitive coupling between neighboring cells in the memory array [12]. Third, threshold voltage instabilities due to repeated cycles of read-only operations can also cause disturbance that results in a significant increase in the number of bit errors [21]. Fourth, it has been shown recently that on partially programmed blocks of MLC NAND that are exposed to a large number of reads before it is finalized in terms of page programming, the remaining pages will exhibit a significant bit error rate increase [14].

An attacker could exploit CCI to alter the information in a victim flash page. This can be achieved by programming an adjacent aggressor page with special data patterns that produce maximum interference to the victim flash page. Due to the nature of the CCI there are only a few cell state transitions that are possible. Specifically, CCI can cause a cell state to transition only to a larger threshold voltage. Using CCI, an attacker can program an aggressor page with a maximum interference pattern to cause uncontrolled random modification to all or different fields of cells of the adjacent victim page in a probabilistic manner.

### 2.3 Flash reliability measures

At the flash controller level, two mechanisms are used to increase reliability: scrambling and error correcting codes (ECC). Because some bit patterns are more likely to cause errors, data is not written as-is, but encoded by a *scrambler* to make such errors less likely [20]. Typically, the data is simple XOR'ed with a bit pattern that is a function of the block address.

Regarding ECC, the data written to a flash page is encoded: redundant bits are added to the user data to ensure that errors present in the binary sequence that is read from Flash can be corrected by decoding. The length of a codeword used in Flash storage systems typically ranges from 512B to 2KB. While the two most widely used codes are Bose–Chaudhuri–Hocquenghem (BCH) codes and low density parity check (LDPC) codes, the precise details of the code design are manufacturer-specific.

## 3 Full-system attack

A full-system attack gaining local privilege escalation that uses a flash weakness needs to tackle multiple challenges at different layers of the storage hierarchy. Namely, from the lowest layer to the highest:

1. Flash chip: cell-to-cell interference.
2. Flash controller: scrambler and ECC bypass.
3. SSD Controller: wear leveling and block placement algorithm.
4. OS: filesystem caching and error detection bypass.
5. User: privilege escalation payload.

Layer 1 is the basis of the attack. As explained in Section 2, previous work has demonstrated that it is possible to cause bit flips in flash pages with specific access patterns, in particular using cell-to-cell interference.

| | Confidentiality | Integrity |
|---|---|---|
| Physical | Power analysis [11], Cold-boot [9] | Evil maid [18], device "rooting" [13] |
| Non-physical | Side channels [4, 6, 7] | Rowhammer [10, 19, 22], this attack |

Table 1: Categorization of hardware-based attacks, with some examples.

The Flash community has long known and studied NAND errors, and, in the context of reliability concern for SSDs, manufacturers designed the flash and SSD controllers to make such errors unlikely to occur in practice. In essence, layers 2 and 3 aim to provide to the OS the abstraction of a block device with no reliability issues, and with perfect isolation of accesses at the addressing granularity of the storage device. As often in systems security, this abstraction can be shown to be leaky. We explain next what attack primitives the attacker may obtain from this leaky abstraction.

The remaining challenge (layers 4 to 5) consists essentially in finding a filesystem based attack vector that leverages this weak attack primitive, which is the main contribution of this paper. The general idea of the attack is to cause corruption of a filesystem data structure and to prepare the filesystem and choose the structure in such a way that the corruption is likely to result in a privilege escalating condition (in this case, creating a SUID-root shell binary).

We have explored multiple possibilities, and present in Section 4 the best attack (in terms of success probability) we found.

## 3.1 Attack primitives

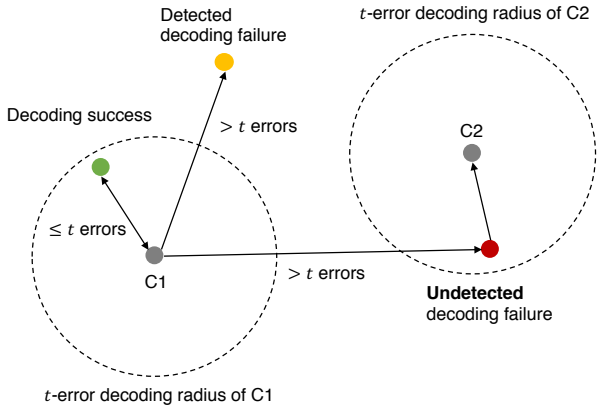


Figure 1: The three possible decoding events that can be induced by injecting errors into a Flash codeword (C1).

The presence of ECC encoding/decoding in the system dictates that the best an attacker can hope for is an uncontrolled random modification of flash page/block primitive. To understand this, let us consider the different decoding events that an attack may be able to cause by injecting raw bit errors into a Flash page. We will assume that the user data has been protected using a BCH code capable of correcting up to $t$ bit errors within its codeword (C1). The value $t$ is typically quite high (e.g., $t = 50$). When introducing errors, there are three possible events that can be induced:

1. Decoding success. This event is guaranteed to occur if the number of errors introduced is less than or equal to $t$.

2. Detected decoding failure. This event will occur if the number of injected errors is strictly greater than $t$ and the resulting binary vector does not fall within the decoding radius of another codeword. Typically, a read failure will be reported by the system which will be handled accordingly by the OS.

3. Undetected decoding failure. This event occurs when the number of injected errors is strictly greater than $t$ and the resulting vector falls within the decoding radius of another codeword (C2). If this event occurs, no read failure will be reported and corrupt data will be returned to the user as if the read was successful.

The three different types of induced decoding events are illustrated in Figure 1. Clearly, a successful attack can only exploit the undetected decoding failure event. Therefore, the attacker must inject enough errors to push the raw read-back pattern into the decoding radius of an incorrect codeword. In this manner, it is possible to cause true data corruption but controlling the pattern of this corruption down to the bit level is not possible. The set of corrupted patterns that can be induced is determined by the number of incorrect codewords that are *reachable* by flipping bits in the written pattern via the CCI mechanism or otherwise.

To contextualize and summarize this, possible attack primitives, from strongest to weakest, may be:

- P1: flip of single bit at controlled location

- P2: uncontrolled flip of single bit within block

- P3: uncontrolled random modification of the block (high number of bits flipped, leading to undetectable decoding failure)

- P4: corrupt one block (read error for this block)

With rowhammer on DRAM primitives P1 or P2 are obtained, but because of the strong ECC protection, the primitives P1 and P2 are highly unlikely to be achievable here. Primitive P4 is highly unlikely to be usable for a privilege escalation exploit (as a side note, the read disturb attack described by Kim et al. [10] may in fact only result in such a primitive, due to ECC). We focus our attack on the third primitive, which is therefore the only one likely to result in a successful attack.

Flash Translation Layer (FTL) operations, like wear leveling and garbage collection (GC) induced flash page writes, as well as the block placement algorithm have to be taken into consideration when implementing an attack. Traditionally, the FTL performs wear leveling in an attempt to equalize the P/E cycle counts across blocks [5]. Since wear leveling is typically an infrequent operation, the attacker can mitigate potential interference to their attack by repeating the attack vector many times. FTL GC write operations, on the other hand, can potentially be a multiple of the user writes [1, 3] in a steady state write workload, and hence comprise a significant fraction of the total flash page writes. In order to avoid interference from GC write operations on an attack, the attacker can wait enough time (typically a few 10s of seconds) for the GC operations to settle before the attack commences. This is possible because SSDs typically prepare a number of empty blocks while idle to sustain a burst of user writes.

In order to maximize parallelism at the Flash chip level, SSDs usually employ block placement algorithms that dynamically map incoming writes to different Flash chips (e.g., in a round-robin fashion). These block placement algorithms can impede an attacker's efforts, since an aggressor page write happening immediately after a victim page write, might not happen at the same Flash block as the victim page. To mitigate this obstacle, an attacker should repeat the aggressor page write enough times to eventually reach the same Flash block as the victim page.

### 3.2 Testbed

Our testbed, depicted in Figure 2, comprises a PCIe Flash development board with FPGA, DRAM, general purpose CPU, and MLC NAND Flash chips. NAND Flash programming, the data scrambler, and the error correction code (ECC) are fully implemented in the FPGA. The FTL runs on the FPGA and the CPU. The development board is connected to an x86-64 server running RHEL 6.7. We have total control over the hardware and software stack

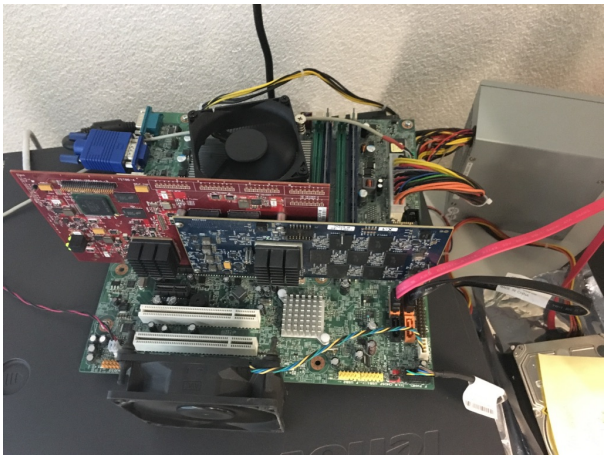

Figure 2: Evaluation testbed: green, red, and blue PCBs are the server motherboard, Gateway and CPU board, and FPGA and NAND Flash board, respectively.

that runs on the development board, as well as the Linux device driver that talks to the device.

The MLC NAND chips used in our testbed have been characterized in prior work in terms of reliability [14, 15, 16]. In the context of this work, the testbed has been used to validate our understanding of MLC NAND flash behavior.

## 4 Filesystem-level attack

We now describe in detail the filesystem-related parts of the attack (layers 4 to 5). A video demonstrating this local privilege escalation attack is available at `https://www.youtube.com/watch?v=Mnzp1p9Nvw0`.

A successful attack needs to satisfy the following constraints:

- **R1**: The data corruption of the target block should have no (or low) probability of resulting in a fatal filesystem error that would stop the attack and require administrator intervention.

- **R2**: The corruption target should be a block that is written often (in order to have a greater chance of success). Ideally, the write should be triggered by the attacker. This is to allow timing this filesystem attack with the layers 1 to 3 of the attack.

- **R3**: The data corruption of the target block should have a sufficient probability of creating an exploitable condition.

- **R4**: The cache should be flushed to force the OS to access corrupted data from the flash disk.

## 4.1 Assumptions and Setup

We implemented the attack on Linux with the ext3 filesystem, mounted with default mount options. The filesystem does not need to be the root filesystem.

We discuss in Section 5 to what extent these assumptions can be relaxed and generalized.

## 4.2 Attack

With the constraints of the attack in mind, we have chosen to target indirect blocks (as victim blocks).

An ext3 indirect block is a filesystem-block-size area containing data block pointers, each 4 byte in size. Therefore, on a 4K-block-size filesystem, an indirect block contains 1024 data block pointers (the data blocks contain file contents).

An indirect block is written (by the kernel filesystem driver) as soon as a file becomes larger than 12 blocks in size: this write is therefore very easy to time and trigger for the attacker.

If the lower layer (layers 1 to 3) of the attack is successful, an indirect block is corrupted, with three possible outcomes for each of the 1024 data pointers in the indirect block.

- The 32-bit data pointer may point to an "interesting block" inside the filesystem (e.g., inode table[1], root ssh private key file, important binary used by root): this case is the only exploitable condition.
- The 32-bit data pointer may point to an "uninteresting block" inside the filesystem, such as data blocks from a file already belonging to the attacking user.
- The 32-bit data pointer may point to a block "outside" the filesystem (i.e.: block number $\geq$ number of blocks on the filesystem). In this case, the corruption is not exploitable. However, luckily, the filesystem kernel driver will only return an error for the corresponding data access: subsequent accesses using the same indirect block, but to another offset inside the file, can still succeed. This behavior of the driver allows the attack to work if *any* of the 1024 corrupted blocks point to an interesting block, greatly increasing the success probability of the attack.

Let's assume the corrupted pointer points to an inode table (exploitable condition). The attacker can then create (or modify) an inode to be root owned and with the SUID-bit on, by simply writing to the victim file (which is under attacker control). The attacker then points the data block pointers of this inode to the data blocks of a shell, and finally elevates privileges to root by executing the victim file, which is now a SUID-root shell (this requires the inode corresponding to the SUID-root shell to be flushed from the cache), as shown in Figure 3.

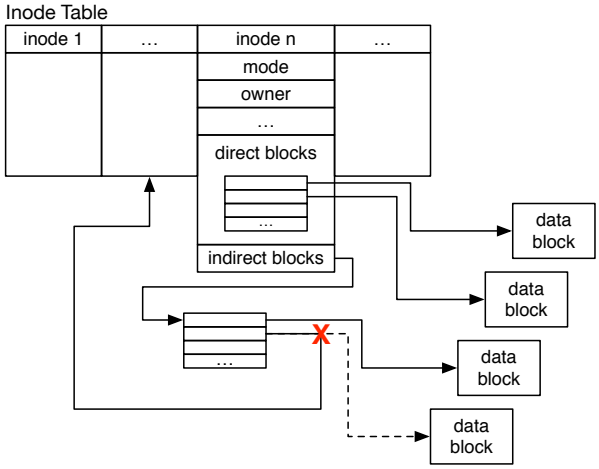


Figure 3: The attack on the filesystem level works by corrupting a data block pointer inside an indirect block, such that instead of an ordinary data block, it points to the inode table. By writing to the attacker-controlled file, we can now use the illegitimate redirection to change the contents of the inode table. Targeting data pointers in the indirect block instead of those inside the inode allows the attacker to increase its success probability, while reducing chances of an irrecoverable error.

We have shown that the **R1** is satisfied because corrupting the indirect block randomly does not result in fatal errors. **R2** is also satisfied because the attacker triggers the write of the indirect block pointer (by simply writing enough bytes into the file). We now show that the **R3** constraint (sufficient probability of creating an exploitable condition) is also satisfied.

The probability $p_1$ of a single data pointer pointing to an interesting block is:

$$p_1 = \frac{no\_interesting\_blocks}{block\_space\_size} \tag{1}$$

And the probability $p_2$ that *any* of the $block\_size/4$ (4 byte block pointers) corrupted pointers point to an interesting block is:

$$p_2 = 1 - (1 - p_1)^{block\_size/4} \tag{2}$$

Assuming only the inode table is "interesting" for the attacker, and a 100 GB filesystem with 4 KB block size, inode ratio of 16384 bytes per inode (default value in `/etc/mke2fs.conf`):

[1] An inode table is an on-disk array of metadata entries (inodes) about files

$$
\begin{aligned}
p_1 &= \frac{no\_inodes \cdot inode\_size}{block\_space\_size \cdot block\_size} \\
&= \frac{(100 \cdot 2^{30}/16384) \cdot 256}{2^{32} \cdot 4096} \\
&= 100/1048576 \approx 0.01\% \\
p_2 &= 1-(1-100/1048576)^{4096/4} \approx 9\%
\end{aligned}
\tag{3}
$$

Thus, assuming that the attacker successfully triggers a one-block data corruption in the indirect inode, a privilege escalation attack will succeed approximately 9% of the time. Therefore, we consider **R3** satisfied. (Note that this probability increases with larger filesystems, larger filesystem block sizes, and smaller inode ratios. Note also that this probability only applies to the filesystem-level of the attack: the full attack's probability is the multiplication of this probability and that of achieving the attack primitive through corruption of the underlying Flash media.)

Finally, we show that caches can be flushed by the attacker (**R4**). The attacker needs to flush caches at two distinct times: To overwrite the inode table contents, and to execute the newly created SUID-root shell. In the first case, the cached data that should be evicted from cache is an indirect block, and in the second case, an inode. The attacker can naturally follow two strategies to this end: a passive or an active one. Passively, the attacker can wait for the filesystem to be remounted (e.g., on reboot), or that enough memory pressure is caused by concurrent activity on the system to evict the indirect block, and later the inode, from cache. The active attacker-triggered route consists in creating memory pressure: the attacker simply uses a program to allocate enough memory, thereby encouraging the OS to evict filesystem metadata from cache. This is the approach we have taken in our demonstration. (The effectiveness of this method may vary depending on VFS and page cache settings, yet, with the default settings on RHEL 6.7 in our attack, the eviction was effective).

## 4.3 Exploitation details

We now discuss some additional details that arise during practical exploitation but are not central to the attack.

First, from the description above it may seem that the attacker needs to know the data block numbers of a root shell. Although this can be looked up by reading from the inode table (through the victim file) and identifying a shell, e.g., by timestamp or file size, this is quite unlikely because the portion of the inode table that is readable is unlikely to contain the root shell. A simpler alternative is that the attacker starts first by creating a copy of a shell into the target file, thereby letting the filesystem create and set the data block pointers, and then sets the SUID bit and makes the file root-owned by directly writing to the inode table as in our demonstration.

Second, the attacker needs to be able to identify when the corruption was successful, i.e., the indirect block pointer points to somewhere inside an inode table as opposed to somewhere else. In practice, this is easy to accomplish because of the structure of inode tables (for instance, the marker for regular files with default permissions is repeated every 256 bytes).

Third, the attacker can overwrite either an existing inode, or a to-be-created file's inode. We can assume we are always in the former case if the attacker has no quotas on the number of files it can create (this is usually the default setting in most Linux distributions). The attacker then creates files until the filesystem runs out of inodes (all inode tables are full), and then after the attack searches in the filesystem (e.g., using `find`) a file that is SUID-root and executes it: this is the way the attack is done in our demo. In the case where the attacker has a quota on the number of files it can create, the attack may still be possible (by deleting files, waiting for the inodes to be reallocated to another user and creating new files until the targeted inode is allocated to the attacker), although we have not attempted this.

Finally, the existing inode might not be accessible to the attacker through the filesystem (e.g., it is in a subdirectory that the attacker cannot access). In practice, this simply means the number of *interesting blocks* is lower than that calculated in equation 3.

## 4.4 Improved attack using double indirect blocks

A very similar attack can be performed by targeting double indirect block corruption instead of indirect block corruption. Although this attack is slightly more difficult to explain, it provides a much higher probability of success and more flexibility to the attacker (full read and write capability over the entire filesystem: note that this still leads to privilege escalation by creating SUID-root files for instance).

Each of the 4-byte pointers in the double indirect block points to an indirect block. Therefore, corrupting a double indirect block can lead to the attacker having full control over the contents of an indirect block. By choosing the pointers in the indirect block, the attacker can read and write arbitrary locations in the filesystem.

The exploitable condition here is that one of the pointers in the double indirect block point to an attacker-owned block. For instance, the attacker can create a very large file on the filesystem (spanning many blocks) to increase the probability that one of the corrupted pointers is attacker-owned. Once the corruption occurs, the attacker can verify that they own the indirect block by filling in the large file

with the address of the filesystem superblock (which is at a fixed block number and in a recognizable format). The attacker can then modify the large file's contents to gain access to any block on the filesystem through reading and writing the target file: the location of the write or read access is controlled by writing to the large file (control of the indirect block contents), and the content of the access is controlled by accessing the target file.

Assuming the attacker can create a 100 GB file (a reasonable assumption nowadays) in a 4 KB block size ext3 filesystem, the success probability of this improved attack is 99.7% (obtained by using equation 2). This is because the number of *interesting blocks* is greatly increased.

# 5 Discussion

## 5.1 Other filesystems

Although many parts of the attack described in Section 4 can be generalized, it remains dependent on filesystems using indirect blocks, such as ext3, ext2, and some versions of the Unix File System (UFS1). The ext4 filesystem uses extents (as NTFS does as well), essentially describing data blocks as a set of `(starting file block, starting disk block, number of blocks)` tuples. Although it is possible to have an "extent block" in ext4 (beyond 3 extents, ext4 starts storing extents in a new block outside of the inode, thereby starting a tree structure), we believe the attack would have a much lower probability of success because of the greater structure of extents (i.e., it is much less likely that a random corruption will not be caught by the ext4 parser: this is compounded by the fact that ext4 typically uses a block number space of 48 bits, instead of ext3's 32 bits). Therefore, whether the attack would still be feasible for other filesystems remains to be seen.

## 5.2 Metadata checksums

In addition, some filesystems such as ZFS and ext4 (but not ext3) optionally allow for metadata checksums to be used. Clearly, filesystem metadata checksums, whether they use cryptographic hashes or not, would significantly lower the success probability of the attack by allowing the kernel filesystem driver to detect that filesystem metadata (e.g., inode or extents) has been corrupted.

Finally, we have experimented with running `e2fsck`, a tool checking (and correcting) ext3 filesystem metadata, after the indirect block corruption occurs. As expected, `e2fsck` successfully detects that filesystem metadata is corrupted, which would allow an administrator to detect this attack. Of course, `e2fsck` is seldom run, and, to thwart the attack, it would need to be run before the attacker gains root privileges and is able to correct the filesystem metadata.

## 5.3 Other attack vectors

Any program (in the broad sense) that accesses the SSD, directly or indirectly, is potentially a target for non-physical integrity attacks on SSDs. In this paper, we have considered filesystems, but other attack vectors could exist.

Unfortunately, because of the restriction given by the weak primitive P3 that the attacker obtains, together with the other constraints (R1 to R4) of the attack, we were unable to find another real-world program that would likely be exploitable for privilege escalation. We cannot, however, exclude this may exist and further research is needed in this area.

Finally, as an analogy to rowhammer and the work of Gruss et al. [8], it is worth considering whether the attack presented in this paper could also be exploited remotely through browser javascript. Because browsers do allow writes and reads to the filesystem (albeit indirectly), through web content local caching, cookies, or use of the HTML5 storage API, it may be feasible to extend the attack vector presented here to remote attacks.

## 5.4 Encryption and Integrity

The use of disk encryption (such as `dm-crypt`) should fundamentally prevent the attack presented here. More precisely, disk encryption would not prevent the attack from proceeding at layers 4 to 5 (because the attacker is overwriting metadata that will be transparently encrypted/decrypted by the OS). However, it would make it highly unlikely that the attacker can gain primitive P3 from weaknesses in layers 1 to 3. Indeed, as explained in Section 3, to bypass the combination of ECC, scrambler, and the restriction that Flash pages can only be programmed in one direction and obtain primitive P3, the attacker needs to control the data written to disk. However, with the use of disk encryption, it would be very difficult for the attacker to be able to do so without knowledge of the encryption key.

As mentioned earlier, metadata integrity would thwart the filesystem attack vector we present. However, other attack vectors, if any realistic application-level one is found in the future, may remain exploitable. We note that current filesystem encryption solutions, with the exception of ZFS, typically do not implement (cryptographic) integrity of file contents because of performance concerns.

## 6 Conclusion

In this paper, we present and tackle the issues that need to be dealt with in order to exploit flash weaknesses in a real-world scenario. On top of this, we show an actual exploit based on flash weaknesses that leverages the ability to corrupt filesystem metadata. In combination these two aspects form a full-system attack vector with an estimated success rate that is reasonable enough to be relevant in practice.

In future work, we plan to address the remaining step of overcoming ECC to complete our implementation of the full-system attack.

## Acknowledgments

We thank our shepherd Per Larsen and anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644412.

## References

[1] Werner Bux and Ilias Iliadis. "Performance of Greedy Garbage Collection in Flash-based Solid-state Drives". In: *Perform. Eval.* 67.11 (2010), pages 1172–1186.

[2] Yu Cai, Augata Ghose, Yixin Luo, Ken Mai, Onur Mutlu, and Erich Haratsch. "Vulnerabilities in MLC NAND Flash Memory Programming: Experimental Analysis, Exploits, and Mitigation Techniques". In: *23rd IEEE International Symposium on High Performance Computer Architecture*. 2017.

[3] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. "Real-time Garbage Collection for Flash-memory Storage Systems of Real-time Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 3.4 (2004), pages 837–863.

[4] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking branch predictors to bypass ASLR". In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE. 2016, pages 1–13.

[5] Eran Gal and Sivan Toledo. "Algorithms and Data Structures for Flash Memories". In: *ACM Computing Surveys* 37.2 (2005), pages 138–163.

[6] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware". In: *Journal of Cryptographic Engineering* (2016), pages 1–27.

[7] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pages 368–379.

[8] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer.Js: A Remote Software-Induced Fault Attack in JavaScript". In: *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*. DIMVA 2016. 2016, pages 300–321. ISBN: 978-3-319-40666-4.

[9] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Cal, Ariel J. Feldman, and Edward W. Felten. "Least we remember: Cold boot attacks on encryption keys". In: *In USENIX Security Symposium*. 2008.

[10] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. 2014, pages 361–372. ISBN: 978-1-4799-4394-4.

[11] Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential power analysis". In: *Advances in cryptology—CRYPTO'99*. Springer. 1999, pages 789–789.

[12] Jae-Duk Lee, Sung-Hoi Hur, and Jung-Dal Choi. "Effects of floating-gate interference on NAND flash memory cell operation". In: *IEEE Electron Device Letters* 23.5 (2002), pages 264–266. ISSN: 0741-3106.

[13] Collin Mulliner and Benjamin Michele. "Read It Twice! A mass-storage-based TOCTTOU attack". In: *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT)*. 2012.

[14] N. Papandreou, T. Parnell, T. Mittelholzer, H. Pozidis, T. Griffin, G. Tressler, T. Fisher, and C. Camp. "Effect of Read Disturb on Incomplete Blocks in MLC NAND Flash Arrays". In: *IEEE 8th International Memory Workshop (IMW)*. 2016, pages 1–4.

[15] Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Thomas Mittelholzer, Evangelos Eleftheriou, Charles Camp, Thomas Griffin, Gary Tressler, and Andrew Walls. "Enhancing the Reliability of MLC NAND Flash Memory Systems by Read Channel Optimization". In: *ACM Transactions on Design Automation of Electronic Systems* 20.4 (2015), 62:1–62:24.

[16] T. Parnell, N. Papandreou, T. Mittelholzer, and H. Pozidis. "Modelling of the threshold voltage distributions of sub-20nm NAND flash memory". In: *2014 IEEE Global Communications Conference*. 2014, pages 2351–2356.

[17] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. "Flip Feng Shui: Hammering a Needle in the Software Stack". In: *USENIX Security Symposium*. 2016, pages 1–18.

[18] Joanna Rutkowska. "Why do I miss Microsoft BitLocker". In: `http://theinvisiblethings.blogspot.ch/2009/01/why-do-i-miss-microsoft-bitlocker.html`. 2009.

[19] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *BlackHat USA*. 2015.

[20] B. Shin, C. Seol, J. S. Chung, and J. J. Kong. "Error control coding and signal processing for flash memories". In: *2012 IEEE International Symposium on Circuits and Systems*. 2012, pages 409–412.

[21] P. Tanduo, L. Cola, S. Testa, M. Menchise, and A. Mervic. "Read disturb in flash memories: reliability case". In: *Microelectronics Reliability* 46.9-11 (2006), pages 1439–1444.

[22] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. 2016, pages 1675–1689. ISBN: 978-1-4503-4139-4.

[23] S. Yamada, Y. Hiura, T. Yamane, K. Amemiya, Y. Ohshima, and K. Yoshikawa. "Degradation mechanism of flash EEPROM programming after program/erase cycles". In: *Proceedings of IEEE International Electron Devices Meeting*. 1993, pages 23–26.