An Analysis of HMB-based SSD Rowhammer

Jonas Juffinger

Graz University of Technology, Graz, Austria

Abstract. Rowhammer has been shown to be an extensive attack vector. In the years since its discovery, numerous exploits have been shown, attacking a wide range of targets from kernels, through web browsers to machine learning models. These attacks were not always mounted from code running on the CPU of a system. Various devices peripheral to the CPU, like GPUs or networks cards can cause Rowhammer bit flips through DMA accesses to the main memory.
In this work, we take a look at solid state drives (SSDs) and if they can be exploited as confused deputies to perform Rowhammer attacks. With the introduction of NVMe, a standardized protocol that allows SSDs to communicate directly over PCIe with the CPU, SSDs have reached performance numbers of a million input/output operations per second. PCIe also enables SSDs to use DMA for direct accesses to the main memory. This lead to the introduction of the host memory buffer (HMB) feature, that allows SSDs to use a small fraction of the host DRAM. We are the first that reverse engineer how different SSDs utilize this host memory buffer and answer the question if the accesses from the SSD to the HMB are a potential attack vector to cause Rowhammer bit flips.
Our analysis of three SSDs shows, that bit flips in the HMB cause the SSDs to lock up, which results in a denial of service or, even worse, data loss. We also show how we can cause frequent accesses from the SSD to the HMB on all three SSDs. On one SSD, we reach 5 000 DRAM accesses per refresh interval. We measure the Rowhammer impact of these accesses and show that they are effectively hammering the DRAM. However, 5 000 DRAM accesses are not enough to cause Rowhammer bit flips, even on modern, highly vulnerable DRAM.

1 Introduction

Since its discovery in 2014 [19], Rowhammer became a large research field, with new and more sophisticated exploits being published every year, escalating privileges [32], hammering from ARM, AMD x86 or RISC-V [12, 26, 35], modifying binaries [8], evading ECC [4], reading secrets [22], attacking the newest generation of mitigations [7, 11], using TRR as a confused deputy [20], finding a new RowPress attack [25] or hammering many banks simultaneously [14]. Researchers and the industry also published and implemented many Rowhammer mitigations [2, 3, 7, 13, 21, 30, 36]. Most attacks hammer the DRAM from code running on the CPU, either from native code or also from interpreted (and JIT compiled) languages like, for example, JavaScript in the web browser.

A few works have shown that Rowhammer attacks are also possible from devices attached to the CPU that can access the DRAM. Frigo et al. [6] hammered on ARM, utilizing the integrated GPU to load texture data from the DRAM to perform the hammer memory accesses. The big advantage of using the GPU instead of the CPU, were smaller, more easily evictable caches. Tatar et al. [34] attacked systems exploiting Remote Direct Memory Access (RDMA). RDMA allows DMA accesses over the network without involvement of the CPU with supported network cards. With networks speeds above 10 Gbit/s, up to 40 Gbit/s, the RDMA accesses were frequent enough to induce bit flips.

Solid state drives (SSDs) are the de facto standard storage medium in every new computer, in particular in mobile ones, like laptops. Highly increased speeds, especially on random accesses, lower power consumption and better physical durability make them superior to HDDs in pretty much every use case where storage space to price is not the number one priority [33]. With the NVMe standard, SSDs are now directly connected to the CPU over PCIe and are reaching over 1 000 000 IOPS with PCIe 4.0. Because of how flash memory storage works internally, logical addresses must be translated to physical addresses for each accesses. The data structure storing these translations, the flash translation layer (FTL), must therefore be quickly accessible for high performance. "Pro"-grade SSDs typically come with a DRAM chip directly next to the SSD controller to store the FTL to maximize performance. However, this is expensive. SSDs supporting the host memory buffer (HMB) feature, can use a part of the main memory to cache FTL translations [28]. For this, the SSD controller request memory from the operating system that it then uses exclusively through direct memory accesses (DMA). This saves cost and achieves acceptable performance.

Zhang et al. [37] identified the FTL, stored in the DRAM, as a potential target for Rowhammer attacks. A bit flip in the correct location of the FTL can remap a page on the SSD, similar to how Seaborn et al. [32] remapped a memory page by flipping a bit in a virtual memory page table. This remapping could give an attacker access to an indirect mapping block of the file system that is inaccessible to unprivileged users. By editing this mapping block, the attacker gains read and write access to the whole file system. They use an emulated SSD from the Linux Foundation's (prior Intel's) Storage Performance Development Kit (SPDK) [24] for their evaluation of this attack.

In this work, we are the first to analyze actual SSDs and their vulnerability to Rowhammer bit flips as well as their potential as confused-deputy attack vectors. By mapping the memory reserved for the HMB and utilizing the IOMMU, we can precisely learn how different SSDs use the HMB during operation. We inject artificial bit flips into the cached FTL mappings in the HMB and observe that the SSD integrity checks the HMB. While this prevents remapping blocks with bit flips, we find that after injecting bit flips, the SSD stops responding to any commands until a complete power cycle is performed. During our experiments we even manage to break one SSD by changing HMB contents while discarding blocks. This would make HMB bit flips an effective denial of service attack.

For a real end-to-end exploit the bit flips must be caused by Rowhammer accesses from the SSD to the HMB. We show, how our understanding of the HMB helps us to achieve frequent HMB accesses by accessing the SSD. The least recently used (LRU) HMB cache replacement policy of the Samsung 990 EVO and Lexar NM790 allows to target specific pages for double sided Rowhammer. However on-chip cache eviction makes these accesses slow, only 700 per refresh interval. The Samsung 980 does not use LRU but a more rigid cache mapping. By accessing a pair of addresses spaced with a specific interval, the Samsung 980 seems to always access the HMB with every read from the SSD. With these address pairs, we achieve up to 5 000 accesses to the DRAM per refresh interval.

As this could be enough to see an impact on bit flip numbers, we evaluate the Rowhammer impact of these HMB accesses from the SSD, with a combined Rowhammer experiment. We prime the DRAM with conventional Rowhammer accesses from software to achieve a minimum number of activations and then fill the rest of the refresh interval with HMB accesses from the SSD. In this experiment we do see a measurable impact from the SSD's accesses, equivalent to the Rowhammer accesses from software. However, with only 78 000 IOPS, the number of SSD accesses is not high enough to induce bit flip on their own.

Even with out attack not working, this work is the first one that looks into the HMB's role in Rowhammer attacks, both as a victim to bit flips and an actively hammering part. We show that bit flips in the HMB can have devastating outcomes, from lost data to broken SSDs. However, the risk of this currently happening is very low. Because SSDs manage their HMB in blocks of at least 4 kB and contain an additional on-chip translation cache, the overhead is too large to achieve memory accesses that are frequent enough to cause Rowhammer bit flips. However, a privileged attacker, could corrupt the HMB through the direct physical map to try to break hardware.

Contributions. In summary, we make the following contributions:

– We perform the first in-depth analysis of the host memory buffer of SSDs, show how it is used and what effects bit flips have on it.
– We exploit the HMB cache replacement policies to cause double-sided Rowhammer accesses and find special SSD access patterns that trigger a maximum amount of HMB accesses.
– We perform a combined Rowhammer experiment with software and SSD accesses, showing that the SSD accesses have measurable impact.
– Finally, we make a compelling argument, why the HMB is currently no Rowhammer security risk.

Outline. In Section 2, we provide background information on SSDs and the host memory buffer (HMB) feature. In Section 3, we give an overview of the attack and threat model. In Section 4, we reverse engineer how SSDs use the HMB and what happens if we inject artificial bit flips. In Section 5, we show how we can cause Rowhammer accesses from the SSDs but that these accesses are not fast enough to cause Rowhammer bit flips. We shortly talk about how the HMB can be disabled in Section 6 and conclude in Section 7.

2 Background

In this section we provide background on Rowhammer, solid state drives and flash memory, the flash translation layer flash memory requires and the host memory buffer that caches recently used translations.

2.1 Rowhammer

Rowhammer is a vulnerability of modern DRAM [19]. DRAM stores bits as charge in an array of capacitors. These capacitors are accessed through a transistor. Frequent accesses to DRAM rows injects electrons into the substrate, where they can travel to neighboring rows' transistors, opening them slightly. This increases the discharge rate of the connected capacitors. If the charge decreases too much before the next refresh happens, the stored bit flips. Rowhammer is a highly researched topic, with a large number of exploits [4, 6–8, 11, 12, 14, 20, 22, 25, 26, 29, 34, 35] as well mitigations [2, 3, 13, 21, 30, 36] shown in the past. For Rowhammer to cause bit flips, enough accesses must be made to the rows neighboring the victim row to cause enough discharge. This limit is called the maximum activation count (MAC). While this limit shrinks with every new DRAM generation, from more than 100 000 on DDR3 to only 10 000 on LPDDR4 [27], more advanced mitigations require more complex and frequent dummy accesses [7, 11].

2.2 Solid State Drives and NAND Flash Memory

Solid state drives (SSDs) are storage devices that use solid-state flash memory instead of spinning disks in hard disk drives (HDDs), to persistently store data. This has the big advantage that it is mechanically more robust and enables much higher random input/output operations per second (IOPS).

In contrast to HDDs, the NAND flash memory cannot be randomly written. The mechanisms to set bits to `1` (erasing) and setting them to `0` (programming) is different. In its empty state NAND memory is erased and stores all `1`'s. To write data to the memory, bits are programmed to `0`. Reprogramming is possible as long as the new data is a subset with only `1` to `0` transitions.

Reading and programming typically happens in sizes of 512 B, 2 048 B or 4 096 B, called a page. However, erasures must happen in blocks of 32, 64 or 128 pages and take considerably longer. These erasures also slightly damage the NAND cells each time. Therefore, flash memory has only a limited number of so called program-erase cycles (P/E cycles). To extend their lifespan, SSDs perform wear-leveling, where the erasures are evenly distributed over all blocks.

2.3 Flash Translation Layer

To efficiently work with the different sizes for erasing and programming data and perform wear leveling, flash memory does not one-to-one map logical addresses

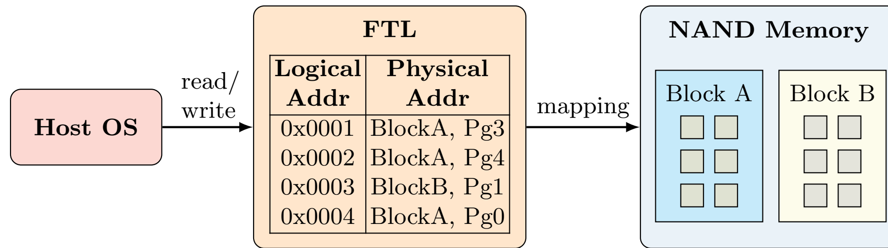


Fig. 1: The flash translation layer translates logical page addresses to physical page addresses. It is required because pages are scattered throughout the flash memory for performance and wear-leveling reasons.

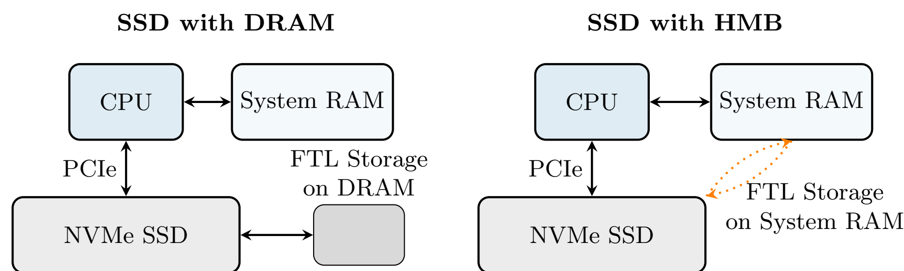


Fig. 2: SSDs with a host memory buffer use a small fraction of the host DRAM to cache recently used FTL translations. This slower but cheaper than a separate DRAM chip on the SSD.

to the actual physical addresses of the NAND memory blocks and pages. Instead, data is scattered all over the flash memory, requiring a translation layer between the logical and physical addresses, called a flash translation layer (FTL). Figure 1 shows the basic concept of the FTL.

Every time the data is read from or written to the SSD, the logical addresses coming from the host must be translated to the physical address with the FTL. Therefore, the latency of FTL accesses has a significant impact on the SSD's performance, especially for random accesses. Therefore, SSDs use different caches to decrease FTL access latency. "Pro"-grade SSDs come with a DRAM next to the SSD controller, large enough to contain the whole FTL. When starting, the SSD copies the whole FTL into the DRAM and only reads it from there, greatly increasing random IOPS. Because this DRAM chip adds cost, a feature called host memory buffer (HMB) was introduced with NVMe revision 1.2 [28].

2.4 Host Memory Buffer (HMB)

NVMe SSDs, being PCIe devices, can use direct memory accesses (DMA) to access the host DRAM without CPU involvement. This is used to enable the host memory buffer (HMB) feature that allows SSDs to use a small part of the

main memory to cache FTL translations [28]. Access to the main memory are slower than accesses to DRAM integrated on the SSD itself but they are still faster than having to access the flash memory for every FTL translation. The HMB is a cache and typically not large enough to hold the whole FTL. HMB support was introduced into the Linux kernel in 2016 [5].

The standard does not mention what the contents of the HMB should be. This is intellectual property of the SSD controller manufacturers and entirely undocumented. There is no research on the HMB yet, except for its performance impact [15–18].

Estimating the circulation of SSDs supporting the HMB feature as difficult, as even SSD manufacturers do not always document HMB support for their SSDs. Of the approximately 20 SSDs we checked, four support the HMB feature.

2.5 Input-Output Memory Management Unit (IOMMU)

A memory management unit in modern CPUs virtually partitions and protects the main memory, that is actually physically shared between all processes. In protected and long mode, page tables are set up by the operating system, to define exactly which process is allowed to access which memory pages [10]. On x86-64 a normal page is 4 kB large, this is the smallest unit the main memory can be partitioned into. The MMU also allows the operating system to track the usage of pages by its process with the *dirty* and *accessed* bits, that are automatically updated by the MMU hardware if a page is written or accessed [10].

With the introduction of hardware assisted virtualization, Intel and AMD not only introduced a second level of pages tables to translate guest physical to host physical addresses but also mechanism to partition and protect the physical memory accessed by DMA capable devices. Intel calls this feature Intel Virtualization Technology for Directed I/O (VT-d) [9] and AMD I/O Virtualization Technology (AMD-Vi) [1]. At the heart of both is a Input-Output Memory Management Unit (IOMMU), that uses very similar page tables than the MMU with a similar feature set.

3 Attack Overview and Threat Model

In this section, we illustrate how a successful attack would look like with all components of the attack working as shown in Figure 3. The basic idea, especially the exploit of the ext4 file system, is based on the work by Zhang et al. [37]. However, they used an SSD simulator with simulated integrated DRAM and no HMB, while we perform our experiments on real SSDs with the HMB feature. The attacker has the permissions to create and read files from a file system on the target SSD. It has no other privileges. By frequently accessing specific files, they create an access pattern to the SSD, that forces the SSD to access the HMB frequently. These accesses to the HMB are frequent enough to cause a bit flip in the FTL. This changes one logical to physical page mapping of the HMB. With some luck, the new mapping either points to a file of interest, for example,

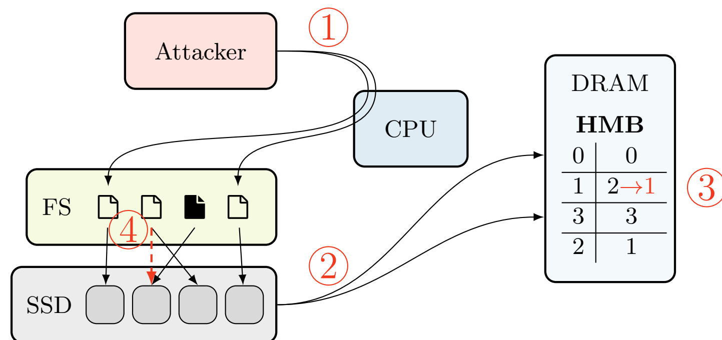


Fig. 3: The complete exploit chain, if every step worked. The attacker accesses specific files on the file system (FS) (1). The SSD has to access the HMB to translate the logical block addresses requested by the file system driver to the physical addresses of the NAND blocks (2). These accesses to the HMB cause enough Rowhammer accesses to flip a bit in the translation stored in the HMB, changing one mapping (3). This changed mapping allows the attacker to access the block where the secret file (black) is stored at through another file, either directly (4) or by getting access to a mapping block of the file system.

Table 1: The SSDs used in our experiments.

SSD	PCIe Revision	Size	HMB Size	IOPS[1]
Samsung 990 EVO	4.0	2 TB	64 MB	680 000
Lexar NM790	4.0	2 TB	40 MB	1 000 000
Samsung 980	3.0	1 TB	64 MB	500 000

[1] Read IOPS with maximum queue depth as advertised by the manufacturer.

containing a secret key or it points to a internal mapping file of the file system. This could give the attacker access to every file on the file system. In this work we show, that neither flipping bits to change an FTL mapping, nor hammering with the SSD works with real SSDs.

4 Reverse Engineering the HMB

In this section, we reverse engineer how SSDs are using their HMB, how bit flips in the HMB memory area influence translations and how more complex HMB modifications can revert FTL mappings. Table 1 shows the SSDs used in this work. All of them use different SSD controllers supporting the HMB feature.

4.1 Tooling

The HMB specification only defines the very basic principles on how the SSD can request host memory from the operating system and how the operating system

```
1 # cat /sys/block/nvme0n1/device/hmb_addresses
2 0 cfc00000 400000
3 1 cf800000 400000
4 ...
5 14 cc400000 400000
6 15 cc000000 400000
```

Listing 1.1: A part of the output of `/sys/block/nvmeXn1/device/hmb_addresses`. The first column is the memory chunk index, the second the I/O virtual address, and the third the length.

```
1 # cat /sys/kernel/debug/iommu/amd/iommu00/0000:01:00.0/page_tables
2 ...
3 0x00000cc000000 | 0x0  0x0  0x6000000101e7b421  0x6000000101e9f221  0x7000000113000021
4 0x00000cc001000 | 0x0  0x0  0x6000000101e7b421  0x6000000101e9f221  0x7000000113001021
5 0x00000cc002000 | 0x0  0x0  0x6000000101e7b421  0x6000000101e9f221  0x7000000113002021
6 0x00000cc003000 | 0x0  0x0  0x6000000101e7b421  0x6000000101e9f221  0x7000000113003021
7 0x00000cc004000 | 0x0  0x0  0x6000000101e7b421  0x6000000101e9f221  0x7000000113004021
8 ...
```

Listing 1.2: A part of the output of `/sys/kernel/debug/iommu/amd/iommu00/0000:01:00.0/page_tables` showing five levels of page table entries.

responds to these requests [28]. The operating system or any other process must not access the memory reserved for the HMB [28]. How the SSD uses the HMB is completely vendor specific and undocumented. In this section we describe the changes to the Linux kernel and our user space tools we use to gain more insight into how the HMB is used.

Linux Kernel Modifications We modify the Linux kernel version 5.15 and utilize the IOMMU to get detailed insight on how different SSD controllers use the HMB.

HMB Mapping. When the Linux kernel reserves memory for the HMB, it does so with the `DMA_ATTR_NO_KERNEL_MAPPING` flag, not mapping the reserved memory because the specification prohibits its access anyway. As we want to know and change the HMB content in our experiments, we add a device attribute to the NVMe driver that prints the physical addresses of the memory chunks reserved for the HMB. The attribute is accessible through the sysfs file `/sys/block/nvmeXn1/device/hmb_addresses`. Listing 1.1 shows the output with the I/O-virtual addresses because the IOMMU it active. The operating system reserves 16 memory chunks, 4 MB each for the 64 MB HMB of the Samsung 990 EVO.

HMB I/O Page Tables. With the IOMMU active, the SSD accesses I/O virtual addresses that are translated to physical addresses on each DMA access. To map the HMB in user space, we also have to translate the I/O virtual to the physical addresses. Additionally, we want access to the I/O page table entry's accessed

(`A`) bits to get detailed information about which pages the SSD accessed. To do this, we need access to the IOMMU page tables. The Intel IOMMU Linux driver already comes with a DebugFS interface to dump all page tables levels for a device at `/sys/kernel/debug/iommu/intel/{PCI_ADDRESS}/page_tables`. For the AMD IOMMU driver we wrote our own in a similar fashion. Listing 1.2 shows a part of the output of the Samsung 990 EVO.

Page Table Entry Accessed Bit. According to Intels IOMMU documentation, the accessed (`A`) and dirty (`D`) bits are always set in page tables entries [9]. However, we did not see the `A` or `D` bits being set on any of our Intel CPUs. The documentation does also not define a register where support for the `A` or `D` bit can be checked or where it can be activated.

AMDs IOMMU implementation can set the `A` and `D` bit if supported by the CPU and activated by the driver [1]. Support is specified by bits 49 (HASup) and 52 (HDSup) in the IOMMU Extended Feature Register and can be activated in bits 47:46 (HADUpdate) in the IOMMU Control Register. We added the check for support and activation to the AMD IOMMU kernel driver. We found that *only* Zen 3 CPUs support the `A` and `D` bit in their I/O page table entries. Neither Zen 2 nor Zen 4 CPUs do. Therefore, we use an AMD Ryzen 7 5800X for all of our experiments in this work.

IOMMU Page Size. The AMD IOMMU supports additional page sizes between the 4 kB, 2 MB and 1 GB of the conventional MMU. This can improve performance, because less page table entries are required to translate other buffer sizes. To get the best resolution for the `A` bit, we change the AMD IOMMU driver to only support and map 4 kB pages. With the Intel IOMMU driver this behavior can be forced with the kernel command-line parameter `intel_iommu=sp_off`.

User Space Tools In user space, we parse the outputs of `/sys/block/nvmeXn1/device/hmb_addresses` and `/sys/kernel/debug/iommu/amd/iommu00/{PCI_ADDRESS}/page_tables` and map the memory for the HMB and the last level page table entries with PTEditor [31]. This gives us read and write access to the HMB content, as well as the page table entries with the `A` bit.

4.2 HMB Usage

To get more insight into how the HMB is used by the different SSDs, we performed many experiments reading and writing to the SSD while observing how the HMB contents change and which pages were accessed. We shortly summarize the most important insights for this work on our three SSDs.

Samsung 990 EVO. The Samsung 990 EVO, like the other two SSDs, uses the HMB only to cache FTL translations. Listing 1.3 shows a small fraction of the HMB content after reading 100000 random pages from the SSD. There is definitely structure recognizable in this data, however, we keep a detailed analysis

```
1 0x3f73000:  30ae30d030ae30c  30ae30f030ae30e  30aa3100036c500  30aa312030aa311
2 0x3f73020:  30ae310030aa313  30ae312030ae311   1eeae9030ae313  30aa315030aa314
3 0x3f73040:  30aa317030aa316  30ae315030ae314  30ae317030ae316  50aa30c001e5677
4 0x3f73080:  50aa30e050aa30d  50ae30c050aa30f  50ae30e050ae30d  3d1ee8b050ae30f
5 0x3f73080:  50aa311050aa310  50aa313050aa312  50ae311050ae310  50ae313050ae312
6 0x3f730a0:  50aa31403f9c162  50aa316050aa315  50ae314050aa317  50ae316050ae315
7 0x3f730c0:  3f97dfc050ae317  70aa30d070aa30c  70aa30f070aa30e  70ae30d070ae30c
8 0x3f730e0:  70ae30f070ae30e  70aa310001ef7f2  70aa312070aa311  70ae310070aa313
```

Listing 1.3: Content at HMB offset 0x3f73000 after reading 100000 random pages from the SSD.

of what it means for future work. The HMB is split into four sets, 16 MB large, each set uses a last-recently used (LRU) cache replacement policy. The "cache-line" size is one page, *i.e.*, 4 kB. Every time a new translation is cached at least 4 kB are written and we also suspect that always at least 4 kB are read from the HMB. The Samsung 990 EVO also contains a small on-chip cache that caches approximately 140 pages of translations for even faster accesses. The SSD is often loading multiple pages from the HMB, this is probably a performance optimization similar to an adjacent line prefetcher inside a CPU.

After not accessing the SSD for around 100 ms the HMB is completely reset. The content is not actually cleared, but when accessing the SSD again, the HMB is filled from the first page of every set again. This makes the HMB accesses very predicable, because they can always be reset by sleeping.

Lexar NM790. The Lexar NM790 uses a LRU HMB eviction policy similar to the Samsung 990 EVO. However, it uses the HMB as one single set. We also only see accesses to one or, most of the time, multiple 4 kB pages. The on-chip cache has a capacity of approximately 20 pages.

Samsung 980. The Samsung 980 also accesses the HMB in blocks of at least 4 kB. However, it does not use LRU HMB eviction. The mapping between the logical addresses and where their translations are stored in the HMB seems to be more rigid. We did not reverse engineer a set function but were able to find "cache thrashing" access patterns, where each access to the SSD, accesses the same HMB page due to this more rigid mapping. We detail and use this pattern for our hammer attempts in Section 5.1.

4.3 Simulating Rowhammer Bit Flips

In this section, we artificially flip bits in the HMB to observe how the SSD reacts. We find that every one of our tested SSDs locks up as soon as it detects the bit flip. We even broke one SSD that we could not get back working.

For this experiment, we map the memory reserved for the HMB writeable and write to it from our user space program. We want to simulate the effects a successful Rowhammer attack would have on the FTL of the SSD. As the

```
1 [   30.920489] nvme nvme2: Device not ready; aborting initialisation, CSTS=0x0
2 [   30.920517] nvme nvme2: Removing after probe failure status: -19
```

Listing 1.4: The kernel log output when the kernel tries to initilize the broken Samsung 980 during boot.

physical location of the HMB is allocated once at boot time, it does not move while the operating system is running. We also found the location of the HMB being the same most of the time, as well as being surrounded by other kernel mappings. This means that an attacker has no way to template the memory, the HMB is later stored at, for viable bit flip locations. Therefore, we randomly flip one or two bits at random locations in the HMB.

When flipping bits in the HMB, the SSD does not directly react. If it does not need the HMB page with the bit flips anymore, it just gets evicted. In this case, the bit flips are never detected and accessing the SSD page corresponding to the HMB page later, loads the unmodified data back into the HMB. An attacker would avoid this in an attack scenario.

SSD Lock Up. By reading the SSD page, corresponding to HMB page with the bit flip, after reading some other SSD pages to evict the on-chip cache, the HMB page with the bit flips is read by the SSD. In this case, all of our tested SSDs *instantly lock up*. We suspect that a checksum of each page in the HMB is stored in the SSD controller. All requests to read or write data are never responded and simply time out. Also other NVMe commands sent with the `nvme-cli` tool are ignored. Resetting the controller or subsystem with with `nvme reset` or `nvme subsystem-reset` did not reverse the lock up. Also rebooting or turning the machine off and, after some time, on again, does *not* reset the SSDs and reverse the lock up. We found that only a real power cycle, unplugging or turning of the PSU of the computer brings the SSDs back. It seems like that even a turned off computer still supplies PCIe devices with power, preventing a reset of the SSDs.

Breaking SSDs. During our experiments we permanently broke one Samsung 980 SSD. With a combination of `blkdiscard` commands and writes to the HMB, the SSD locked up and does not work anymore, even after power cycles. It does not respond to any commands and the kernel gives up initializing it during boot. Listing 1.4 shows the kernel log output. For budgetary reasons, we did not further investigate the exact course of events to reproduce the issue a second time.

Discussion. Our experiments show that the ext4 exploit by Zhang et al. [37], that requires a remapping of flash memory blocks because of bit flips, is not possible when hammering the HMB. However, already a single bit flip in the HMB requires a power cycle of the whole machine. This can lead to a denial of service in the best case and a corrupt system or data loss in the worst case if the

```
1 0x228f500: 0xb4691cd680300010 0x39ad3a4735a548e6 0x548f6b4691ed6852 0xd6a523dad3a47b5a
2            0xb4691cd68630000c 0x39ad3a4735a548e6 0x548f6b4691ed6852 0xd6a523dad3a47b5a
3 0x3a111e0: 0x08de003a1613149e 0x1da9144607d70994 0x02851bba1159092f 0x027f04530e801f2d
4            0x08de003a1613149e 0x1da914460bc40994 0x02851bba1159092f 0x027f04530e801f2d
```

Listing 1.5: Changes in the HMB after writing to one SSD page. Unchanged bytes are in gray, changed bytes are in red (previous value) and green (new value). Four bytes change in total in two different pages. Resetting these bytes to their previous value maps the previously used block.

operating system is not able to flush dirty data to the disk. In a more advanced attack, an attacker could try to coerce the operating system to send frequent discard commands to trim the SSD. The attacker hammers the SSD at the same time to break the SSD, causing physical damage on a modern computer only from software. This would also be possible for an attacker that already gained elevated privileges and can write to the HMB through the direct physical map.

4.4 “Unwriting” Data

By carefully changing data in the HMB, we managed to “unwrite” data on the Samsung 980 SSD. To do this, we stored the HMB content before writing to one SSD page. This write changes four bytes in the HMB, as shown in Listing 1.5. Changing these bytes to any other values locks up the SSD, except for one change. If we reset the changed bytes to their previous values, the FTL mapping is reset to the old value. Reading the previously written SSD page returns the *old* content of the page. As the HMB content is never written back to the SSD, this change is not persistent. After the eviction of the modified HMB page, reading the SSD page again returns the new content.

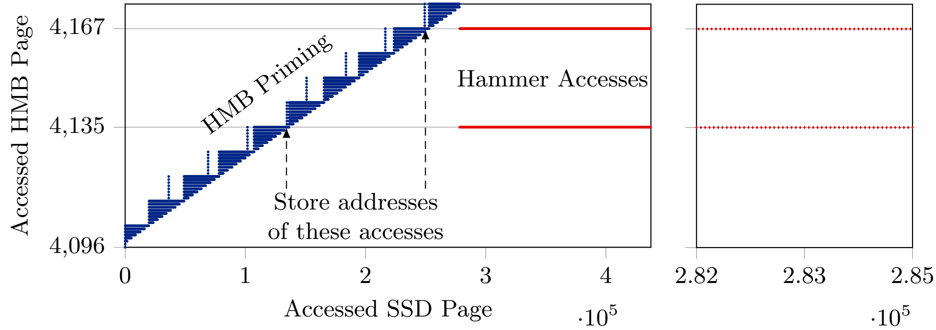
Being able to perform a replay attack like this, conflicts with our hypothesis that the pages cached in the HMB are simply protected with a checksum that is stored in the SSD controller. A few bits of the changed data could also be checksum bits. However, as changing data like this is far outside the reach of Rowhammer bit flips, we did not further investigate this effect.

5 HMB Rowhammer

While actual privilege escalation with the ext4 exploit [37] with bit flips in the HMB is very unlikely due to all SSDs instantly locking up, a denial of service attack or maybe even breaking SSDs is possible with flips in the HMB. However, we show that the maximum achievable accesses from the SSD to the HMB are not high enough to flip bits, even on highly vulnerable DRAM.

5.1 Generating Hammer Accesses

In this section, we show how we can cause targeted Rowhammer accesses from the different SSDs to the HMB. All SSDs behave differently and require different



(a) The blue dots show the accesses to prime the HMB with translations. The stored SSD page addresses are then used for the hammer accesses.

(b) Zoomed in plot on the hammering phase.

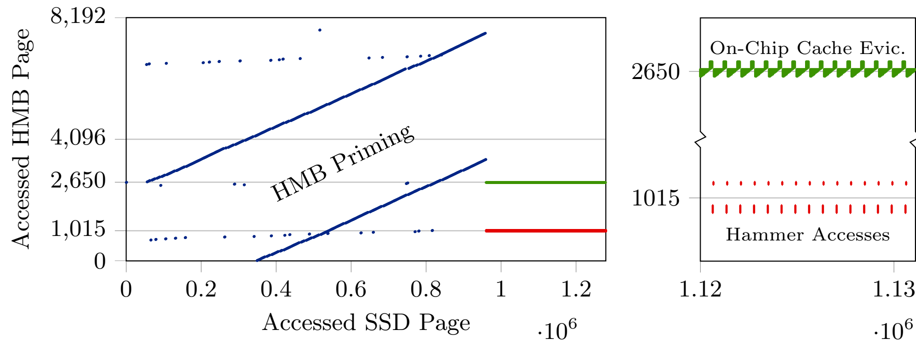
Fig. 4: HMB priming and hammering on the Samsung 990 EVO.

accesses patterns to frequently access the HMB. We achieve the fastest accesses on the Samsung 980, with 4 992 hammer accesses per refresh interval.

Samsung 990 EVO. This SSD uses a LRU cache eviction policy and four 4 sets for its HMB. The LRU algorithm makes it possible to prime the cache with translations and then later accesses the pages with the cached translations to get double-sided Rowhammer accesses.

Figure 4a shows the cache priming and hammering. In the priming phase, we sequentially access contiguous pages on the SSD, storing where they caused an HMB accesses. The SSD shows interesting behavior, visible as a stair pattern. At some accesses to the SSD, it accesses up to 8 pages in the HMB. We expected that these 8 pages are loaded into the on-chip cache and the SSD would not accesses the HMB for them again. But instead, we see that after accessing 32 pages on the SSD, the SSD accesses the HMB again, loading the *same* pages. It seems like the on-chip cache caches the data for a only 32 accesses. We could not think of a reason why this would be of an advantage. The SSD then slowly reduces the number of pages it accesses until it accesses only a single page Then it accesses the next 8 HMB pages. With this slope being constant and it always starting the same way, we compute exactly how many SSD pages we have to access to prime the HMB for double-sided Rowhammer accesses.

For this example, we assume very simple DRAM addressing functions, where two pages, 32 pages apart, hammer the page exactly in the middle. If we want to hammer HMB page n, we first prime the cache by accessing $(n+16) \cdot 3640$ pages on the SSD, e.g., from the beginning. 3640 is the slope of the priming-slope. Then we only have to access the SSD pages $(n-16) \cdot 3640$ and $(n+16) \cdot 3640$ to cause the pattern shown in Figure 4b. While we figured out these steps with our modified kernel to get the accessed pages with the IOMMU, they are always repeatable and can, therefore, also be executed by an unprivileged attacker.



(a) The blue dots show the accesses to prime the HMB with translations, the red dots the hammer accesses and the green dots the on-chip cache eviction accesses.

(b) Zoomed in plot on the hammering phase.

Fig. 5: HMB priming and hammering on the Lexar NM790.

This access pattern seems to hit some internal bottleneck of the SSD. Performing it asynchronously with maximum queue depth, reduces the IOPS to 135 000. This is one firth of the advertised 680 000. This results in less than 300 hammer accesses to the two aggressor rows per 64 ms refresh interval.

$$\frac{135\,000\,\text{IOPS} \cdot 64\,\text{ms}}{31\,\text{Dummy Accesses}} = 280\,\text{Hammer Accesses}$$

Checking Asynchronous HMB Accesses. Figure 4 was measured with synchronous SSD accesses to get exactly which access to the SSD cause which HMB accesses. With asynchronous SSD accesses with large queue depths, it is impossible to check the `A` bit in the IOMMU page tables for each SSD accesses. Instead, in another thread we periodically reset all the `A` bits in all page table entries mapping the HMB, flush the IOTLB and check them again without any added delay in-between. It takes only 26 µs to check all `A` bits after flushing the IOTLB. We constantly see the same HMB pages being accessed during this small interval, giving us high confidence that the SSD is actually accessing the HMB also when we perform the SSD accesses with large queue depths.

Lexar NM790. The Lexar SSD also uses a LRU replacement policy. However, it never accesses HMB pages that are cached on the on-chip cache. Therefore, we have to perform on-chip cache eviction. We also use sequential SSD accesses to prime the HMB and then access two specific SSD pages to hammer the HMB and in-between accesses to other SSD pages to evict the on-chip cache. Figure 5a shows the priming and hammer accesses. Because the Lexar SSD's HMB state cannot be "reset" by simply sleeping, the HMB priming does not start from zero. This gives us less control over what exactly is hammered and also how many pages are accesses for the hammer accesses. As shown in Figure 5b multiple

Fig. 6: The hammer accesses from the Samsung 980. The first access to the SSD causes many HMB accesses. Then it takes 360 accesses for the SSD to "settle in". Afterward, every single access to the SSD, causes the SSD to access pages around index 3 606 and 14 854 of the HMB.

pages are accesses for each aggressor. We find that the on-chip cache is evicted after accessing 20 other SSD pages and achieve up to 717 double-sided hammer accesses per refresh interval.

$$\frac{112000\,\text{IOPS} \cdot 64\,\text{ms}}{10\,\text{Eviction Accesses}} = 717\,\text{Hammer Accesses}$$

Samsung 980. The Samsung 980 does not use an LRU cache eviction policy. Neither in the HMB nor in the on-chip cache. This helps us to achieve our fastest Rowhammer accesses of almost 5 000 hammer accesses per refresh interval. On the Samsung 980 it is possible to access specific pages of the SSD, so that each access to the SSD makes the SSD access the HMB, as shown in Figure 6. Each access to the SSD causes the "hammer"-accesses in red to the HMB.

To do this, we take two addresses, 37 200 kB apart, and access them alternately, adding 1 200 kB after each accesses. After 32 accesses, we start again with the initial two addresses. Table 2 shows the pattern of SSD accesses and the HMB accesses they cause. This pattern leads to the SSDs accessing the same pages in the HMB on each SSD page access, perfect for hammering.

However, these special patterns reduce the IOPS to around 78 000 from the advertised maximum of 500 000. But even then, if every one of these accesses to the SSD cause an access to the HMB, we still reach close to 5000 hammer accesses per refresh interval.

$$78000\,\text{IOPS} \cdot 64\,\text{ms} = 4992\,\text{Hammer Accesses}$$

While our thread that checks the accessed bits asynchronously is a good indicator of the HMB accesses from the SSD, we use a combined Rowhammer experiment to measure the real impact of the HMB accesses on Rowhammer bit flips.

Table 2: Accessing these pages on the SSD causes these HMB accesses. $PageA$ and $PageB$ are 37 200 kB apart and the $step$-size is 1 200 kB. After $PageB + 15 \cdot step$ we accesses $PageA$ again. The accesses to $512 - 519$ every 8th time can be seen in Figure 6 as the blue dots at the bottom.

Accessed SSD Page	Accessed HMB Pages
$PageA$	$512 - 519$, $3606 - 3607$, $14848 - 14855$
$PageB$	$3605 - 3607$, $14854 - 14855$
$PageA + step$	$3606 - 3607$, $14854 - 14855$
$PageB + step$	$3605 - 3607$, $14854 - 14855$
$PageA + 2 \cdot step$	$3606 - 3607$, $14854 - 14855$
$PageB + 2 \cdot step$	$3605 - 3607$, $14854 - 14855$
...	
$PageA + 4 \cdot step$	$512 - 519$, $3606 - 3607$, $14848 - 14855$
$PageB + 4 \cdot step$	$3605 - 3607$, $14854 - 14855$
...	
$PageA + 15 \cdot step$	$3606 - 3607$, $14854 - 14855$
$PageB + 15 \cdot step$	$3605 - 3607$, $14854 - 14855$

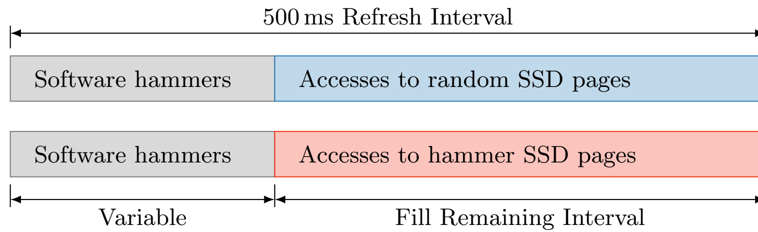


Fig. 7: For the combined Rowhammer experiment, two rows are first hammered from software. Within the same refresh interval, the same rows are then hammered through the SSD's HMB accesses.

5.2 Combined Rowhammer

To measure the real Rowhammer impact of the Samsung 980's access to the HMB, we perform a combined Rowhammer experiment. The idea is to combine Rowhammer accesses from software with accesses from the SSD, as shown in Figure 7. During each refresh interval, we first perform software accesses to get the DRAM victim row to an access count where it starts to flip. Then, we fill the rest of the refresh interval with accesses from the SSD to the HMB. By comparing the number of bit flips we get with the targeted hammer SSD accesses we found in Section 5.1 against the number of bit flips with random SSD accesses, we get the impact of the targeted SSD accesses on the bit flip count.

For this experiment, we use an Intel Core i9-9900K for two reasons. It is well known how to hammer on older generation Intel CPUs, and it supports changing the DRAM refresh interval t_{REFI} in the BIOS. The disadvantage of the Intel CPU is that it does not set the `A` bit in IOMMU page table entries.

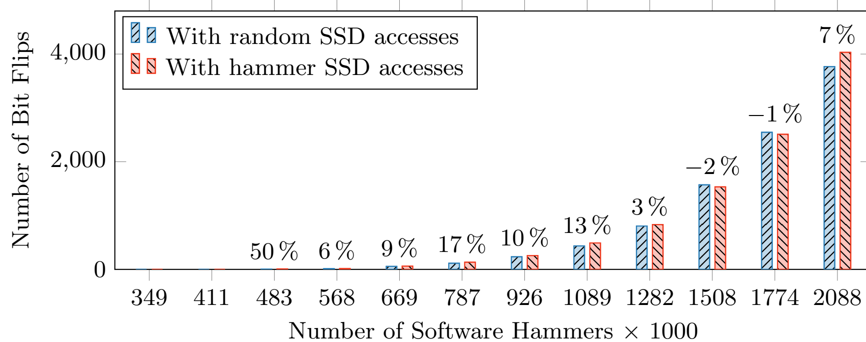


Fig. 8: The number of bit flips when combining Rowhammer accesses from software with accesses to the HMB. The HMB accesses have no measurable impact on the bit flip count. At high conventional hammer access counts it even reduces the bit flip count significantly.

Getting HMB Access Info on Intel CPUs. There is one other, much more tedious way, to get the accessed HMB pages without the `A` bit. We just want to ensure that the SSD still accesses the same HMB pages as it did with the AMD CPU. We unmap the HMB pages in question by clearing the present bit in the corresponding IOMMU page table entries. Now, when we perform our hammer SSD accesses, the SSD tries to access the unmapped HMB page. The IOMMU raises an interrupt and the handler prints the violating access to the kernel log, where we can see that the SSD tried to access the expected HMB page. However, the SSD's DMA access fails and it locks up until a full power cycle. Nevertheless, we confirmed that the SSD accesses the HMB pages found in Section 5.1.

Experimental Setup. We maximized the refresh interval to approximately 500 ms from the default 64 ms. This increases the number of hammer accesses achievable with the Samsung 980 to up to 38 500 per refresh interval.

We want to sweep the victim row through DRAM, to get as much bit flip data as possible. To achieve this, without having to find more SSD page patterns that cause HMB accesses at different offsets, we use the IOMMU to remap pages 3 606 and 3 607. We always map the pages to physical locations that perform double-sided Rowhammer. When remapping the HMB pages, we also have to copy the content to the new physical location to keep the HMB consistent. This creates a short race condition, where the SSD crashes sometimes. We set up a service that automatically power cycles the computer when this happens.

After remapping the aggressor HMB pages, we start with 2 088 000 software accesses, filling the rest with SSD accesses. The 2 088 000 take up approximately half of the refresh interval. We hammer the same victim row 6 times, randomized 3 times with random SSD accesses and 3 times with our targeted SSD accesses, always resetting the victim data in-between, by writing the inverted binary data from the above aggressor row into it. Then, we reduce the software accesses by

15 % and hammer the same victim row again. We repeat this 12 times, down to 349 000 software accesses. Finally, we go to the next victim row, remap the aggressor HMB pages, and start again with 2 088 000 software accesses.

Results. Figure 8 shows the results from the combined Rowhammer experiment. The bar hight shows the number of bit flips with either random accesses to the SSD or the hammer accesses to the SSD. The percentage above the bars shows the change with the hammer accesses. We measured 19 432 bit flips in total.

There is a small but clear increase in the number of bit flips with the SSD's hammer accesses. Over all numbers of software hammers, the increase in bit flips is 3.4 %, from 9 555 to 9 877. This shows that the SSD's accesses to the HMB do actually hammer the DRAM.

Hammer Effectiveness. We can compute the effectiveness of the SSD's HMB hammers. Using the bit flip counts with random SSD accesses from Figure 8, we can calculate that an increase of 1 000 software accesses, increases the bit flip count by 1.8 on average. Now we can calculate by how much the SSD HMB hammers increase the bit flip count. For this we look at the results with 1 089k software accesses.

1 089k software accesses take approximately 130 ms, leaving 370 ms for the SSD accesses to the HMB within one refresh interval. The SSD hammers with 78 000 IOPS for these 370 ms, causing HMB 28 860 accesses.

At 1 089k software accesses, we measured 436 bit flips with random SSD accesses and 491 with hammer SSD accesses, an increase of 55 bit flips or 13 %. Knowing that 1 000 software accesses increases the bit flip count by 1.8, we calculate the equivalence of SSD accesses to the HMB to software accesses.

$$\frac{1000 \text{ software accesses}}{1.8 \text{ bit flips}} \cdot 55 \text{ bit flips} = 30556 \text{ equivalent hammer accesses}$$

The SSD causes 28 860 HMB accesses, meaning that the SSD's HMB accesses are *equivalent* to software acesses.

Discussion. This show clearly that the SSD can hammer the DRAM through the HMB very effectively. However, the IOPS and, therefore, the access count is not high enough to induce bit flips without additional software accesses. There is currently no threat model where an unprivileged attacker could get access to the memory where the HMB is mapped to perform additional software accesses. Furthermore, we used the IOMMU to map two HMB pages to exact locations to perform double-sided Rowhammer.

6 Mitigations

In this section, we show that the HMB can easily be disabled on Linux to mitigate these attacks and discuss the impact of state-of-the-art Rowhammer mitigations on HMB Rowhammer.

Disabling the HMB. The HMB is specified as an optional feature for the SSD: "The controller shall function properly without host memory resources." [28]. The only downside is a reduced performance, as shown by previous work [15–18]. On Linux the HMB can be turned off for all SSDs attached to the system with the kernel command-line parameter `max_host_mem_size_mb=0` [23].

Existing Rowhammer Mitigations. Modern DDR4 and DDR5 DRAM comes with on-die Rowhammer mitigations like TRR. To evade these mitigations, complicated access patterns are required that "confuse" the TRR mechanism so it tracks and refresh other dummy rows instead of the actual victim row [7, 11]. To perform these access patterns, high timing accuracy as well as synchronization with refreshes is important. Additionally, Kang et al. [14] showed that modern Intel CPUs also contain a undocumented Rowhammer mitigation that can only be evaded by hammering multiple banks simultaneously. This means, that while only a few thousand accesses to the actual attacker rows are sufficient on modern DRAM, overall, the whole access pattern to actually cause bit flips on modern system still consists of many more accesses. Currently, we do not see a possibility to achieve the high number of required accesses, to the many different rows, synchronization with refreshes, and hammering multiple banks simultaneously when using HMB hammer.

7 Conclusion

Rowhammer is a long standing security vulnerability. Although, there is countless research on attacks, only very few works looked into the risk coming from devices peripheral to the CPU that can also access the DRAM. This work is the first one that analyzes the SSD as a potential victim or confused deputy attacker of a Rowhammer attack. With the introduction of the HMB feature, NVMe SSD can access the DRAM through DMA to cache FTL translations. We show that bit flips in the HMB are unlikely to lead to privilege escalation attacks but can force a system to power cycle or even break hardware. We also show how our tested SSDs can be coerced into accessing the HMB with a high frequency. On the Samsung 980 we achieve 5000 accesses to the HMB per 64 ms refresh interval. In the combined Rowhammer experiment, together with Rowhammer accesses from software, we see a clear impact of the HMB accesses from the SSD on the bit flip count. This shows that, in theory, an SSD could hammer the DRAM but it very unlikely on modern DRAM with active Rowhammer mitigations.

8 Acknowledgments

I thank Daniel Gruss, Fabian Rauscher and the anonymous reviewers for their valuable feedback on this work. This research is supported in part by the European Research Council (ERC project FSSec 101076409), and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85). Additional funding

was provided by generous gifts from Red Hat and Google. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. AMD: I/O Virtualization Technology (IOMMU) Specification, rev 3.09 (2023)
2. Aweke, Z.B., Yitbarek, S.F., Qiao, R., Das, R., Hicks, M., Oren, Y., Austin, T.: ANVIL: Software-based protection against next-generation Rowhammer attacks. ACM SIGPLAN Notices **51**, 743–755 (2016)
3. Brasser, F., Davi, L., Gens, D., Liebchen, C., Sadeghi, A.R.: CAn't Touch This: Software-only Mitigation against Rowhammer Attacks targeting Kernel Memory. In: USENIX Security (2017)
4. Cojocar, L., Razavi, K., Giuffrida, C., Bos, H.: Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In: S&P (2019)
5. Dawn, A.: [PATCH] NVMe:Support for Host Memory Buffer(HMB) (2016), `https://lore.kernel.org/linux-nvme/20160615104800.GC32253@infradead.org/T/`
6. Frigo, P., Giuffrida, C., Bos, H., Razavi, K.: Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU. In: S&P (2018)
7. Frigo, P., Vannacci, E., Hassan, H., van der Veen, V., Mutlu, O., Giuffrida, C., Bos, H., Razavi, K.: TRRespass: Exploiting the Many Sides of Target Row Refresh. In: S&P (2020)
8. Gruss, D., Lipp, M., Schwarz, M., Genkin, D., Juffinger, J., O'Connell, S., Schoechl, W., Yarom, Y.: Another Flip in the Wall of Rowhammer Defenses. In: S&P (2018)
9. Intel: Virtualization Technology for Directed I/O, rev 4.0 (2022)
10. Intel: Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide (2024)
11. Jattke, P., van der Veen, V., Frigo, P., Gunter, S., Razavi, K.: BLACKSMITH: Rowhammering in the Frequency Domain. In: S&P (11 2021)
12. Jattke, P., Wipfli, M., Solt, F., Marazzi, M., Bölcskei, M., Razavi, K.: ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms. In: USENIX Security (2024)
13. Juffinger, J., Lamster, L., Kogler, A., Eichlseder, M., Lipp, M., Gruss, D.: CSI: Rowhammer - Cryptographic Security and Integrity against Rowhammer. In: S&P (2023)
14. Kang, I., Wang, W., Kim, J., van Schaik, S., Tobah, Y., Genkin, D., Kwong, A., Yarom, Y.: Sledgehammer: Amplifying rowhammer via bank-level parallelism. In: USENIX Security (2024)
15. Kim, K.S., Kim, T.S.: Performance Evaluation of HMB-Supported DRAM-Less NVMe SSDs. KIPS Transactions on Computer and Communication Systems (2019)
16. Kim, K., Kim, S., Kim, T.: HMB-I/O: Fast Track for Handling Urgent I/Os in Nonvolatile Memory Express Solid-State Drives. Applied Sciences (2020)
17. Kim, K., Kim, T.: HMB in DRAM-less NVMe SSDs: Their usage and effects on performance. PloS one (2020)
18. Kim, K., Lee, E., Kim, T.: HMB-SSD: Framework for Efficient Exploiting of the Host Memory Buffer in the NVMe SSD. IEEE Access (2019)
19. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In: ISCA (2014)

20. Kogler, A., Juffinger, J., Qazi, S., Kim, Y., Lipp, M., Boichat, N., Shiu, E., Nissler, M., Gruss, D.: Half-Double: Hammering From the Next Row Over. In: USENIX Security (2022)
21. Konoth, R.K., Oliverio, M., Tatar, A., Andriesse, D., Bos, H., Giuffrida, C., Razavi, K.: ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks. In: USENIX OSDI (2018)
22. Kwong, A., Genkin, D., Gruss, D., Yarom, Y.: RAMBleed: Reading Bits in Memory Without Accessing Them. In: S&P (2020)
23. Linux: /drivers/nvme/host/pci.c (2024), https://elixir.bootlin.com/linux/v6.11/source/drivers/nvme/host/pci.c#L55
24. Linux Foundation: Storage Performance Development Kit (SPDK) (2025), https://spdk.io/
25. Luo, H., Olgun, A., Yağlıkçı, A.G., Tuğrul, Y.C., Rhyner, S., Cavlak, M.B., Lindegger, J., Sadrosadati, M., Mutlu, O.: RowPress: Amplifying Read Disturbance in Modern DRAM Chips. In: ISCA (2023)
26. Marazzi, M., Razavi, K.: RISC-H: Rowhammer Attacks on RISC-V. In: Workshop on DRAM Security (DRAMSec) (2024)
27. Mutlu, O., Olgun, A., Yağlıkcı, A.G.: Fundamentally Understanding and Solving RowHammer. In: Asia and South Pacific Design Automation Conference (2023)
28. NVM Express, Inc: NVM Express, rev 1.2.1 (2016)
29. de Ridder, F., Frigo, P., Vannacci, E., Bos, H., Giuffrida, C., Razavi, K.: SMASH: Synchronized Many-sided Rowhammer Attacks From JavaScript. In: USENIX Security (2021)
30. Saxena, A., Saileshwar, G., Juffinger, J., Kogler, A., Gruss, D., Qureshi, M.: PT-Guard: Integrity-Protected Page Tables to Defend Against Breakthrough Rowhammer Attacks. In: DSN (2023)
31. Schwarz, M., Lipp, M., Canella, C.: misc0110/PTEditor: A small library to modify all page-table levels of all processes from user space for x86_64 and ARMv8 (2018), https://github.com/misc0110/PTEditor
32. Seaborn, M.: Exploiting the DRAM rowhammer bug to gain kernel privileges (3 2015), http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html
33. Shilov, A.: SSDs Outsell HDDs in Unit Sales 3:2: 99 Million Vs. 64 Million in Q1 (2021), https://www.tomshardware.com/news/ssd-market-shares-q1-2021-trendfocus
34. Tatar, A., Krishnan, R., Athanasopoulos, E., Giuffrida, C., Bos, H., Razavi, K.: Throwhammer: Rowhammer Attacks over the Network and Defenses. In: USENIX ATC (2018)
35. van der Veen, V., Fratantonio, Y., Lindorfer, M., Gruss, D., Maurice, C., Vigna, G., Bos, H., Razavi, K., Giuffrida, C.: Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In: CCS (2016)
36. Yaglikci, A.G., Patel, M., Kim, J.S., Azizi, R., Olgun, A., Orosa, L., Hassan, H., Park, J., Kanellopoulos, K., Shahroodi, T., Ghose, S., Mutlu, O.: BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In: HPCA (2021)
37. Zhang, T., Pismenny, B., Porter, D.E., Tsafrir, D., Zuck, A.: Rowhammering Storage Devices. In: ACM Workshop on Hot Topics in Storage and File Systems (2021)