The HMB Timing Side Channel: Exploiting the SSD's Host Memory Buffer

Jonas Juffinger, Hannes Weissteiner, Thomas Steinbauer, and Daniel Gruss

Graz University of Technology, Graz, Austria

Abstract. Over the past three decades, cache side channels evolved from specialized attacks on cryptographic implementations to generic techniques (e.g., Flush+Reload and Page Cache Attacks) on general-purpose operations. During the last decade, SSDs became the de facto standard persistent storage, where capacity is not the highest priority.
In this paper, we present a novel cache side channel, targeting the host-memory buffer (HMB) used by mid-range SSDs to cache translations from logical page addresses to physical page addresses.
We demonstrate that, compared to page cache attacks, our attacks are significantly faster as we can evict reliably in only 22 ms. Consequently, we propose a hybrid attack, using the slow page cache eviction as little as possible and using the HMB side channel for our main attack. We evaluate the HMB side channel in four practical attacks: First, we evaluate the capacity of the HMB side channel in a covert channel scenario, achieving up to 8.3 kbit/s channel capacity. Second, we demonstrate a UI redress attack using the HMB side channel, where the fake UI element covers the real one within 100 ms. Third, given that multiple pages from different security contexts are translated through the same HMB entry, we demonstrate blind templating attacks, that allow to spy on accesses to arbitrary other files whose translation is co-located in the same HMB entry. We use this to demonstrate a cross-VM covert channel and a remote side channel where an unprivileged process without network access exfiltrates data to a remote system over the network, through the HMB side channel by using an nginx web server as a confused deputy.

1 Introduction

The performance of modern systems is fundamentally limited by memory latency. Caches play a central role in modern systems to alleviate this bottleneck. However, while they reduce the latency of accesses to recently or frequently accessed memory, this latency difference can also be exploited in so-called side-channel attacks, at the beginning, mainly on cryptographic implementations [1, 10]. Later, generic techniques like Prime+Probe [18] or Flush+Reload [25] enabled cache side-channel attacks on general-purpose computations [5, 6].

Despite the growing popularity and the increasing performance of SSDs, they have received relatively little attention in terms of side-channel research. Liu et al. [14] studied the now-discontinued Intel Optane, which has several caches introducing distinct timing differences. Two works focused on SSD covert channels with attacker-controllable FPGAs reaching < 1 bit/s [4, 24]. Recently, Juffinger et al. [9] demonstrated that contention on commodity SSD can be exploited to build covert channels with over 1 kbit/s transmission rate and website fingerprinting with over 90 % accuracy. While page cache attacks also leak spatial information and can be faster [6], they were partially mitigated recently [22], reducing the temporal resolution to one measurement within several seconds due to the high cost of eviction, e.g., of a multi-gigabyte page cache. In concurrent work, we discussed the negative result of performing a Rowhammer attack through the HMB and the effect on bit flips on the HMB [8].

In this paper, we present a novel cache side channel, targeting the host-memory buffer (HMB) [17] used by mid-range SSDs to cache translations from logical to physical page addresses. We investigate four different SSD models supporting the HMB feature and reverse-engineer how these SSDs use it. Our investigations show that the HMB operates either full- or set-associatively, translating contiguous blocks of multiple megabytes of memory. We demonstrate how an attacker can observe the HMB state and how they can evict the HMB through accesses to arbitrary files. As a result, we obtain a side channel with a spatial resolution of a few megabytes, low compared to the 4 kB of page cache attacks [6].

However, our HMB attack is significantly faster as we can evict the HMB in only 22 ms or less. Additionally, we demonstrate how we can use the unprivileged `FIEMAP` ioctl system call to effectively increase the spatial granularity of a few megabytes to the 4 kB of page cache side channels by using the page cache to our advantage. Consequently, we propose a hybrid attack, using the slow page cache eviction as little as possible and the HMB side channel for our main attack.

We evaluate the HMB side channel in four practical attacks: First, we evaluate the capacity of the HMB side channel with an inter-process covert channel, achieving up to 8.3 kbit/s. Second, we demonstrate that we can monitor disk accesses to files on the SSD. We exploit this side channel in a UI redress attack using the HMB side channel within 100 ms. The goal of a UI redress attack is to detect when a program starts and then cover it with a fake UI element to, e.g., steal an entered password. Even without any sharing of files or memory, we can exploit that multiple SSD pages from different security contexts are translated through the same HMB entry in blind templating attacks. Third, we show a covert channel between virtual machines with up to 7.1 bit/s. Finally, we demonstrate a remote side channel where an unprivileged process without network access exfiltrates data to a remote system over the network with up to 8.9 bit/s, by exploiting nginx as a confused deputy.

In summary, the **contributions** are as follows:

- We are the first to analyze the host memory buffer for potential side channels and reverse engineer the behavior of different implementations.
- We show that differences in the access latency leak information about recently accessed pages that can be measured with very high accuracy.
- We present a novel blind templating attack that also works if the attacker has no read rights to the target files.

In Section 2, we provide background information on cache timing side channels and SSDs. In Section 3, we describe the HMB side channel and reverse-engineer how the SSDs use the main memory and how the HMB can be evicted. In Section 4, we evaluate the HMB side channel in attacks on accessible files, showing that yields a significantly higher performance than current page cache attacks. In Section 5, we demonstrate that even without accessible files an attacker can mount a blind templating attack using the HMB side channel. We discuss countermeasures in Section 6. We conclude in Section 7.

2 Background

In this section, we provide background on cache timing side channels, SSDs and flash memory, the flash translation layer and host memory buffer.

Cache Timing Side Channels. Caches are used in computers to store data that is likely to be accessed soon, to decrease the delay of these accesses. They are both, faster and smaller, than the main memory they cache data from. Due to this smaller size, they cannot store all the data from the main memory but only a small subset at any given time.

Caches introduce execution timing differences based on the current cache state. This effect is exploited in cache timing side-channel attacks like Prime+Probe [13, 18] or Flush+Reload [25]. While these techniques have mainly been shown on CPU caches, they also can be applied to other caches [6, 23]. Gruss et al. [6] demonstrated the that flush- and eviction-based attacks on the OS page cache are possible. The page cache buffers data from the disk, in blocks of pages. However, as also reported by Schwarzl et al. [22], the interfaces exploited by Gruss et al. [6] are now more restricted.

Solid State Drives and NAND Flash Memory. Solid state drives (SSDs) are persistent storage devices that use NAND flash memory to store data. They are faster than hard disk drives (HDDs), the number of random input output operations per second (IOPS) is magnitudes higher.

NAND flash memory restricts how data can be written. While data can always be read, settings bits to `1` (erasing) uses a different process than setting bits to `0` (programming). Erased memory stores all `1`'s, data is then written by setting bits to `0`. Typically, the read and programming size is 512 B, 2 048 B or 4 096 B, called a page. Erasures happen in blocks of 32, 64 or 128 pages and take much longer. Flash memory has only a limited number of program-erase cycles. To wear out all blocks equally, the SSD controller performs wear-leveling.

Flash Translation Layer. Wear-leveling and other performance optimizations cause the data on an SSD to be scattered. Therefore, a persistent translation layer is required that translates the logical page addresses, the ones seen by the operating system, to the physical page addresses of actual storage locations.

Table 1. The SSDs used in our experiments.

SSD	Model	PCIe Revision	Size	HMB Size
$\mathcal{A}$	Samsung 990 EVO	4.0	2 TB	64 MB
$\mathcal{B}$	Lexar NM790	4.0	2 TB	40 MB
$\mathcal{C}$	Samsung 980	3.0	1 TB	64 MB
$\mathcal{D}$	Western Digital Blue SN550	3.0	1 TB	32 MB

If the FTL was only stored in the flash memory, every read would incur an additional read. Therefore, SSDs employ different caches for their FTL. Budget SSDs only have a small on-chip cache in the SSD controller. "Pro"-level SSDs contain a DRAM chip that is large enough to hold the entire FTL. Mid-range SSDs can use a feature called Host Memory Buffer as a trade-off.

Host Memory Buffer (HMB). HMB is an NVMe feature that allows SSDs to request main memory from the operating system [17]. The SSD can then access the HMB through direct memory accesses (DMA) over PCIe and cache parts of the FTL there. DMA accesses to the main memory over PCIe are slower than accesses to an integrated DRAM but faster than accessing the flash memory for each FTL entry. The HMB is not large enough to store the whole FTL.

HMB Prevalence. Overall, the HMB feature is not uncommon. We used techpowerup's SSD database [3] containing 869 SSDs from 109 manufacturers to understand the prevalence of the HMB feature in SSDs. Of these 869 SSDs, 694 use the PCIe interface. 37 % or 255 of all PCIe SSDs do not have a DRAM. Of these DRAM-less PCIe SSDs, 97.6 % or 249 support the HMB feature. This shows that the feature is very prevalent in DRAM-less SSDs, as it enables a performance gain at almost no cost.

FIEMAP Ioctl. The file extent map (`FIEMAP`) system call allows unprivileged processes to retrieve the "physical" block addresses of all fragments of a file [12]. In this case, "physical" correspond to the logical block addresses of the SSD as seen by the operating system that are then translated to the real physical addresses using the FTL.

3 The HMB Side Channel

The HMB caches recently used logical to physical translations of the SSD to reduce the access latency to the corresponding pages. We show that access latency differences are measurable and give an attacker valuable information about recently accessed pages on the SSD. In this section, we reverse-engineer the basic functionality of the HMB of four different SSDs and show that the HMB introduces a timing side channel that is observable from user space. Table 1 shows

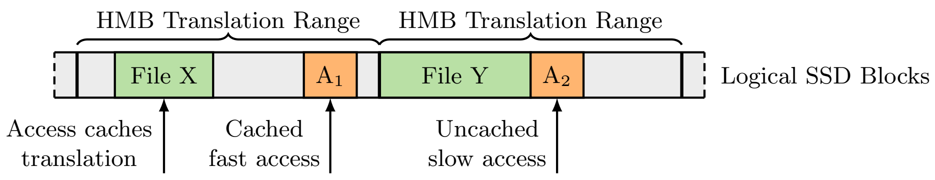


Fig. 1. Working principle of the HMB. An access to file X caches all logical-to-physical translations within a specific storage range in the HMB, including the translation for the page containing file fragment A_1. Therefore, a subsequent access to X or A_1 is faster than an access to Y or A_2.

the SSDs we used for our experiments throughout the paper. Figure 1 shows the basic working principle of the HMB side channel. It shows an SSD's logical storage space as seen by the operating system and its file system driver. After accessing file X, the logical to physical translation to the SSD pages that contain file X but also to the file fragment A_1 are cached in the HMB. This decreases the subsequent access time to X and A_1 while the access to the uncached pages containing file Y or fragment A_2 are slower.

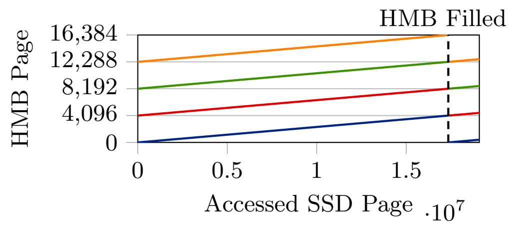
3.1 Reverse Engineering HMB Usage

In this section, we reverse engineer how the different SSDs use the HMB to cache their translations as shown in Figure 2. We use the IOMMU of a AMD Zen3 Ryzen 7 5800X to observe the accesses from the SSD to the HMB with 4 kB page granularity, similar to Juffinger et al. [8].

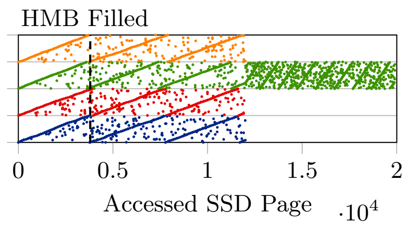
Samsung 990 EVO. The Samsung 990 EVO splits the 64 MB HMB into four independent 16 MB cache sets, as shown in Figure 2a. After accessing 66 GB of data, or 17 million pages, sequentially, the HMB is filled with translations. Afterwards, the SSD controller starts replacing the least recently used (LRU) translations. Bits `14` and `15` of the logical block address select a set.

A lot less *random* accesses are required to fill the HMB, as shown in as Figure 2b. This is because a single access also prefetches many translations of surrounding pages, filling the HMB. There are also seemingly random HMB accesses in-between the linear patterns. These HMB accesses come from accesses to addresses within an already cached HMB translation range. To get size of these prefetches, we build an experiment where we access an addresses and then measure the cache state of a neighboring addresses with increasing distance. This shows that up to 15 360 pages, *i.e.*, translations to 60 MB of data, are cached at once. After 12 000 accesses in Figure 2b, we only access addresses of set 2 by fixing bits `14` and `15` of the logical block address to `10`.

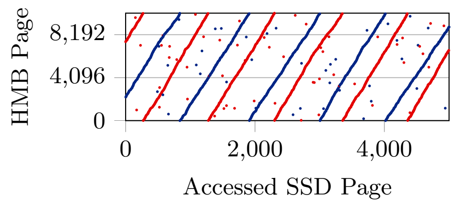
When the SSD is not accessed for 100 ms, it resets the HMB state. Afterwards, each cache set is again filled from the first page and all previously cached translations are not used anymore. The SSD seems to enter a low power state after 100 ms but we cannot think of a reason why this resets the HMB state.



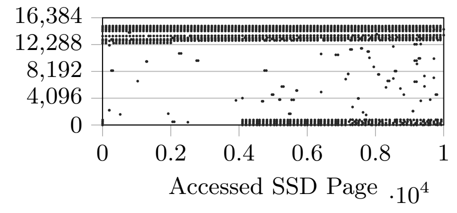
a. $\mathcal{A}$: Sequential SSD accesses causes a linear pattern of HMB accesses. The HMB is split into four sets.



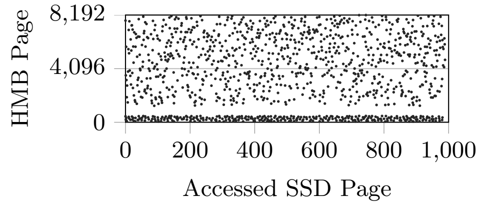
b. $\mathcal{A}$: Random SSD accesses show the linear pattern plus "random" accesses to the HMB.



c. $\mathcal{B}$: Sequential and random (shown) SSD accesses show two LRU patterns.



d. $\mathcal{C}$: Sequential and random (shown) SSD accesses show a pattern to specific offsets at the start and end of the HMB range.



e. $\mathcal{D}$: Sequential and random (shown) SSD accesses show no particular pattern.

Fig. 2. The accesses from the SSD to the HMB caused by accesses to the SSD. The four SSDs behave very differently.

Lexar NM790. The Lexar NM790 also uses LRU cache eviction, as shown in Figure 2c. It also uses two cache-sets, but differently than the Samsung 990 EVO. Both cache sets can occupy every page of the HMB, but they seem to use an individual LRU counter. Bit 25 of the logical SSD address defines the set. We find in Section 3.3 that accessing only addresses of a single set greatly increases variance in the access timings. So it seems like these two sets enable some sort of load balancing in the SSD.

Samsung 980. The Samsung 980 does not show a clear usage pattern like the Samsung 990 EVO or Lexar NM790, neither ith sequential nor random accesses, as shown in Figure 2d. Most parts of the HMB were only used very sparsely. The Samsung 980 also resets the HMB state after 100 ms.

WD Blue SN550. This SSD splits the HMB into two distinct parts, as shown in Figure 2e. Every translation consists of two HMB accesses. One to the lower

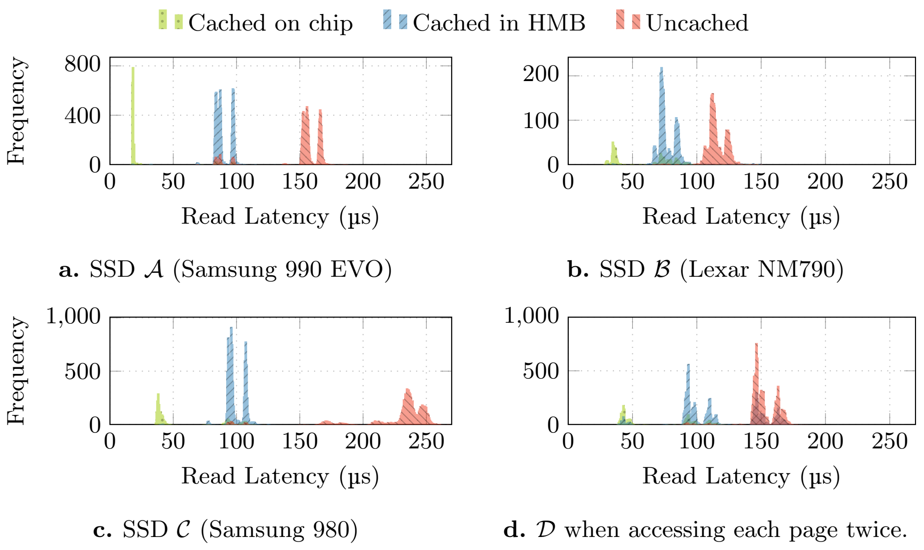


Fig. 3. The timing differences between a translation cached on-chip, cached in the HMB and an uncached translation are clearly distinguishable on our SSDs.

part, between HMB pages 48 and 488 and the other to the upper part above HMB page 1255. We think that the lower part contains a cache directory to store which translations are stored in the upper part of the HMB. The other SSDs seems to store this directory in the SSD controller, requiring more memory.

3.2 Cache Timing Differences

In this section, we show that the timing differences caused by the HMB cache are measurable and clearly distinguishable by user space applications, see Figure 3. We start by measuring with direct access to the block device and verify that we can also distinguish the cache levels when accessing files as an unprivileged user.

For this experiment, we first “reset” the HMB state by accessing 50 000 random pages of the SSD. Then we access 500 additional random pages and store their addresses. As the translations to these addresses are now cached in the HMB, we put them in our “HMB cached” set. The last N_C of those, we put in the “on-chip” set. With the sets initialized, we randomly choose between accessing a random, very likely uncached page, a random page from the HMB set or a random page from the “on-chip” set. We repeat these access 200 times and continuously update the sets of HMB cached translations by adding all new performed uncached accesses and the “on-chip”-set by always keeping the last N_C accesses in this set. The size N_C is only an estimate for the on-chip cache size because knowledge of the exact size is not required for our attacks.

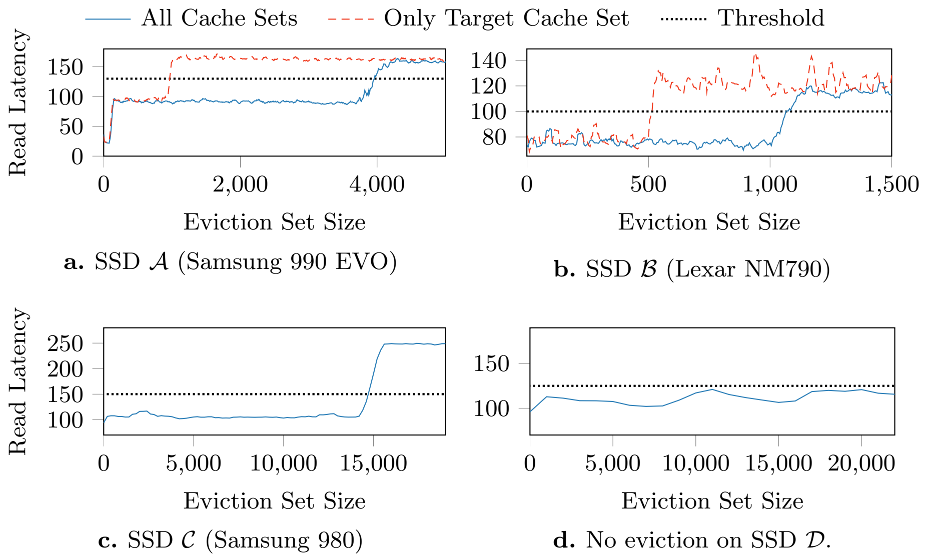


Fig. 4. Eviction with accesses to random pages of the SSD.

Samsung 990 EVO. Figure 3a shows the timing histogram. The timings are separated by 50 µs with a threshold between uncached and HMB cached accesses of 125 µs. The on-chip cache has an approximate size of $N_C = 150$ entries.

Lexar NM790. Figure 3b shows the timing histogram. The cached accesses are still clearly separated from the uncached accesses with a threshold of 100 µs. The on-chip cache is smaller with an approximate size of only $N_C = 40$ entries.

Samsung 980. Figure 3c shows the timing histogram. The timings are again very clearly separated with a threshold of 130 µs. The Samsung 980 has a very small on-chip cache $N_C \approx 5$.

WD Blue SN550. 3d show the timing histograms when accessing every page twice. Almost no translations are cached, when performing the experiment like on the other SSDs by only accessing each page once. With two accesses, translations get cached but there are still many uncached timings for recently accessed pages.

Unprivileged File Access. While we measured the histograms shown in Figure 3 by directly accessing the block device, we verified that we measure the same timings when accessing files as an unprivileged user. As already shown by Juffinger et al. [9], the overhead from the file system is negligible.

3.3 HMB Eviction

Measuring the HMB state of a target file is destructive. After the read, the translation is cached and has to be evicted again quickly. In this experiment, we find the minimum number of accesses required to evict the HMB. To do this, we read a random page from the SSD, caching the translation in the HMB. Then we access an increasing number of pages, either randomly or sequentially. For the sequential accesses, we do not access directly neighboring pages but pages with a stride length of one HMB translation range. We use `io_uring` to submit the reads asynchronously to maximize IOPS. Finally, we access the randomly selected page again and measure the access time. Based on the thresholds from Section 3.2, we know if the page was evicted or not.

Samsung 990 EVO. Figure 4a shows eviction with random accesses to all or one cache set. When accessing random addresses, 4 000 accesses are required to evict a translation from the HMB. This takes on average 45 ms. However, due to the four sets, a more targeted eviction is possible. By only accessing addresses that fall in the same set as the target, with bits `14` and `15` matching, the eviction only requires 1 000 accesses and takes on average 22 ms.

When accessing the SSD sequentially, 17 million accesses are required to evict the HMB. Our hypothesis is, that sequential accesses are optimized to not pollute the HMB with translations only used once. Translations evicted from the on-chip cache could, e.g., be dropped instead of being moved to the HMB.

Lexar NM790. Figure 4b shows eviction with random accesses to all or one cache set. It takes at least 1 150 random disk accesses, which takes on average 11 ms. When only accessing the same cache set as the target is in, by fixing bit `25`, the eviction time is halved to 530 accesses (6 ms). However, when only accessing addresses from the same cache set, there is a significantly higher variance in the access timings: $\sigma = 29.5$ vs $\sigma = 15.7$ when accessing both sets, $n = 1000$.

Samsung 980. On the Samsung 980 it takes around 15 000 random accesses (340 ms) to evict a translation from the HMB, as shown in Figure 4c. Sequential accesses, even for over 20 s, only very infrequently evict the target from the HMB.

WD Blue SN550. We did not manage to reliably and quickly evict a translation from the HMB on the WD Blue SN550 with neither sequential nor random accesses, as shown in Figure 4d. Even when accessing more than a 100 000 pages, multiples times each, for over a second, our target page translations was still fast. Our hypothesis is, that the SN550 can store more detailed translations due to a bigger directory. We show in Section 3.1 that the SN550 accesses the lower 2 MB of the HMB for every translation, hinting its usage as a directory. While other SSDs store their directory on the controller, they only store translations of large blocks of addresses in the HMB. Storing translations of only small blocks, maybe even in page granularity, makes eviction very tedious.

4 Attacks On Accessible Files

In this section, we show a covert channel and a UI redress attack. In this threat model, the attacker has read permissions for the file they want to observe. This means that the attacker can measure the translation cache state of the file they want to observe directly by reading it. We show a covert channel between processes that share access to files in `/usr` and other shared directories in Section 4.1. The covert channel reaches up to 8.3 kbit/s. In Section 4.2, we show a UI redress attack that can detect when `pkexec` is started with a high accuracy.

As we could not reliably evict the HMB of the WD Blue SN550, we are not further evaluating our attacks with it.

Cache Eviction. The big advantage of the HMB side channel over the page cache side-channel is that evicting the HMB is a magnitude faster while causing no memory pressure. Gruss et al. [6] managed to evict the page cache in 149 ms, while it took Juffinger et al. [9] 347 ms with lower memory pressure. Evicting the HMB takes only 22 ms or less on the Samsung 990 EVO and Lexar NM790. This enables a higher sampling rate for more accurate attacks and smaller blind spots. For even faster eviction on the Samsung 990 EVO or Lexar, the attacker needs to know the cache set of the victim file. For this, we use the unprivileged `FIEMAP` ioctl to get the physical mappings of the file on the disk.

If the target we want to observe is in the page cache, we have to evict it. This is once at the beginning and then only after the event we monitor, like the execution of a binary, actually happens. We perform our HMB timing measurements with `O_DIRECT` to not load the target into the page cache.

Preventing False Positives. Due to the design of the HMB, multiple translations are cached together in what we call the HMB translation range. While we exploit this in blind attacks in Section 5, they can cause false positive measurements. A false positive happens if not our target file is accessed and its translation therefore cached in the HMB, but any other file that is in the same HMB translation range. We can reduce the chance of this happening with the *help* of the page cache.

Again, with the unprivileged `FIEMAP` ioctl, we can get all files we have read permissions for that are in the same HMB translation range. On a not strongly fragmented file system, the chance that we have the permissions to read neighboring files is high. If our target is, for example, a binary or library file, they probably reside next to other binary or library files which are typically readable on Linux. After learning which files could cause fault-positive measurements, we read them to load them into the page cache. Therefore, every other program reading any of these files does not access the SSD but gets the file served from the page cache. We then add these files to our page cache eviction set, so they are not evicted when we have to evict our target file from the page cache.

4.1 Covert Channel Across Processes

In this section, we show a covert channel across processes that can transmit data with up to 8.3 kbit/s, as shown in Table 2.

Table 2. All covert-channel results.

Threat Model	SSD	# Co-Locations	Transmission Rate	Bit Error Rate	Channel Capacity
Process (Section 4.1)	$\mathcal{A}$	256	4.4 kbit/s	3.9 %	3.3 kbit/s
	$\mathcal{B}$	512	8.8 kbit/s	0.7 %	8.3 kbit/s
	$\mathcal{C}$	256	594 bit/s	1.4 %	532 bit/s
Virtual Machines (Section 5.1)	$\mathcal{A}$	1	7.1 bit/s	0 %	7.1 bit/s
	$\mathcal{B}$	1	1.9 bit/s	0 %	1.9 bit/s
	$\mathcal{C}$	1	1.7 bit/s	0 %	1.7 bit/s
Remote (Section 5.2)	$\mathcal{A}$	1	10 bit/s	1.5 %	8.9 bit/s
	$\mathcal{B}$	1	4 bit/s	3.0 %	3.2 bit/s
	$\mathcal{C}$	1	2.7 bit/s	0.9 %	2.5 bit/s

Threat Model. Two processes are on the same machine without any means of communication. They have access to a shared set of files, e.g., in in `/bin`, `/lib` or `/usr` and a precise clock source like `rdtsc` or `clock_gettime`. If no process has access to enough files to evict the HMB, at least one of the processes must be able to create a file for HMB eviction. No other privileges are required.

Implementation. The basic idea is to find a set of files both processes have access to and use each file to transmit a bit of the message. This enables for the transmission of multiple bits per time slice and HMB eviction. For this to work, two requirements for these files must be met: First, each file used for the transmission must be in its own HMB translation range, so they are all independent of each other. Second, the two processes must agree on the files used for the transmission and the order of these files.

File Independence. If two files used to transmit individual bits were in the same HMB translation range, setting one bit would always also cache and, therefore, set the other bit. We use the unprivileged `FIEMAP` ioctl to get the physical block location of each file on the disk. The sender and receiver can then select files that are spaced out enough on the disk. Large or fragmented files can also be used to transmit multiple bits if they span multiple HMB translation ranges. We do not perform *false positive prevention* for our covert channel.

File Selection and Order. Both, the sender and receiver must use the same files in the same order to transmit the message. To achieve this, we only use files that are world readable, if a file is readable due to owner or group permissions, it is not used. To use the files in the correct order, we lexicographically order the full file paths and then randomize them with a seed known to the sender and receiver. The randomization ensures that we do not perform too many sequential accesses during transmission which could trigger HMB optimizations of the SSD.

Transmission. The sender and receiver open the files with the `O_DIRECT` flag to evade the page cache. With both processes sharing a precise clock source, we can use a time-sliced approach like prior work [4, 9, 13, 19, 24]. This means that the transmission is divided into time slices known to the sender and receiver.

In the first time slice, the sender accesses all files corresponding to `1`-bits, to cache their translation. We use *two* files per bit. In the next time slice, the receiver accesses the files and records slow timings from *both* files as a `0` and one or two fast timings as a `1`. The third time slice is used by the sender to evict the HMB. Sender and receiver can swap their roles for bi-directional communication.

We use `io_uring` for asynchronous accesses for sending, receiving, and eviction. While asynchronous accesses are considerably faster, issuing them too fast can cause contention in the SSD [9], slowing down the accesses and interfering with our cache timing measurements. Therefore, we find the optimal delay between the measurement (receive) accesses for all three SSDs. We also had to add a shorter delay between the sending accesses.

We record the raw transmission rate and bit-error rate to compute the true channel capacity, the maximum transmittable information over a noisy channel. A bit-error ratio of 0 or 1 corresponds to a perfect transmission; 0.5 means no information was transmitted. We use the binary symmetric channel model to compute the true channel capacity T as $T = C \cdot (1 + (1-p) \cdot \log_2(1-p) + p \cdot \log_2(p))$ where C is the raw transmission rate and p the bit-error rate.

Results. All results are shown in Table 2. On the Samsung 990 EVO we transmit 128 bits using 256 files, reaching a raw transmission rate of 4.4 kbit/s with a 3.9 % bit error rate, resulting in a channel capacity of 3.3 kbit/s. We use a 10 µs delay between the send accesses and 50 µs between the measurement accesses. We only access SSD pages in a single set to make the eviction four times faster.

On the Lexar NM790 we got an even lower error rate while doubling the number of files used for the transmission to 512. With them, we transmit 256 bits per time slice for a raw transmission rate of 8.8 kbit/s with a bit error rate of 0.7 %. This results in a channel capacity of 8.3 kbit/s. The raw transmission rate is not only increased by the larger number of files used, but also by the smaller required delays between the send and receive accesses, 3 µs and 20 µs respectively. This enables us to access all 512 files for sending and receiving, in the same time, we can send and receive 256 files on the Samsung 990 EVO. We use SSD pages from both cache sets. Using a single set increased the variance of the timing measurements and the error rate, decreasing the channel capacity.

The very slow eviction that takes over 350 ms on the Samsung 980, makes its covert channel considerably slower. Accessing the 256 files for sending and receiving takes up only 15 % of each time slice. However, the bit error rate is also very low with only 1.4 %. With a raw transmission rate of 594 bit/s, this results in a channel capacity of 532 bit/s.

4.2 Authentication UI Redress Attack

In this section, we show that we can detect when a file is accessed on the disk with high accuracy. We use this for a user-interface redress attack [2, 6, 20]. In an UI redress attack, the attacker waits for a program to be started to then draw their own fake window over the genuine one. In our example, we detect the start of `pkexec` to steal the password of the victim user. `pkexec` displays a password promt to execute GUI applications with super user privileges. For this to work well, the latency between the genuine drawn and the fake window drawn over it must be small. The short eviction time of our HMB side channel enables this low latency detection. It takes at most 100 ms to detect the start of `pkexec`.

False Positive Prevention. To prevent false positives, which would be strange for the victim and could make them suspicious, we load the other files in the same HMB translation range in the page cache. To test the viability of this approach, we use a privileged user to find all co-located files for both a 2 MB and a 60 MB HMB translation range. In the 2 MB surrounding `/usr/bin/pkexec` find that all 15 co-located files are in `/usr/bin`. All of these files are readable by a user without elevated privileges. In the 60 MB range, we find 1 969 files. Constantly keeping all of these files in the page cache is time consuming. We therefore exclude files that are unlikely to be read, like man files for other locales (e.g., `/usr/share/man/id/man1/`), firmware images in `/lib/firmware/`, or `/lib/x86_64-linux-gnu/fwupd-plugins-5/`. This leaves us with 793 files, overall 46.5 MB large, to keep in the page cache.

Some of these files are log files. Keeping them in the page cache does not prevent disk accesses if the file is written. To prevent false positives from writes, we check the last modification date of all of our 793 files when we measure a HMB cache hit. If one or more of the files where written in the last few seconds, we count the cache hit as a false positive.

Implementation. We implement a page cache eviction algorithm similar to Gruss et al. [6]. However, we only have to perform page cache eviction once at the beginning of the monitoring to evict `pkexec`. We, additionally, perform page cache eviction every minute because our monitoring could miss an execution of `pkexec` due to the blind spot during HMB eviction.

In the monitoring loop, we start by sleeping for 100 ms, during this time `pkexec` could be executed. This means that it takes us at most 100 ms to detect the execution of `pkexec`. During this sleep interval the Samsung SSDs require periodic reads to keep the HMB active and preserve the HMB state, we perform one 4 kB read each millisecond. Then, we perform our measurement by reading the first page of `pkexec`. Our attacks also works by reading a co-located file like `ping`. It could be less suspicious if a program opens `ping` instead of `pkexec`.

Performing the measurement caches the translation in the HMB. This creates a blind spot until the HMB is evicted. For the Samsung 990 EVO and Lexar NM790 we know the correct cache set, because we know the physical block address from `FIEMAP`. This enables quick HMB eviction.

Results. On the Samsung 990 EVO SSD the eviction takes on average 23.2 ms ($n = 1000$, $\sigma = 0.18$ ms). This means that one whole measurement window takes 100 ms + 23.2 ms = 123.2 ms. In this window, we are blind for 23.2 ms or 18.8 %. This is a trade-off between the relative size of the blind spot and the time it takes at most to detect the execution of `pkexec`. On the Lexar NM790, the eviction takes only 6 ms resulting in a blind spot of only 6 %. We chose a sleep duration of 100 ms, because this means that with a 30 fps screen we are at most 3 frames too late to draw our window over the `pkexec` window.

Because our attack process sleeps a lot and also block device accesses are not CPU intensive it, does not cause easily noticeable CPU usage. The attack causes on average 3.3 % CPU usage and 32 MB/s disk usage.

The slow eviction on the Samsung 980 takes approximately as long as page cache eviction. This defeats the main advantage of the HMB side channel over the page cache side channel, making the page cache an easier and more portable attack target.

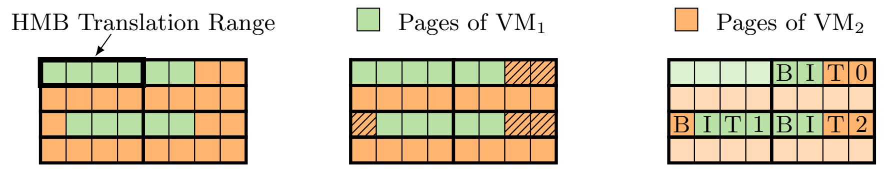
5 Blind Attacks

In this section, we show how attacks can be performed if the attacker does not have read permissions for the files it wants to observe. Because the attacker cannot read the file it cannot use the `FIEMAP` ioctl to get the exact mappings of the file on the disk. Therefore, we fall back to a templating approach, where the attacker learns which parts of the attacker file are in the same HMB translation range as the victim files. We first show a covert channel between two virtual machines in Section 5.1, reaching up to 7.1 bit/s. In Section 5.2, we show how an attacker can exploit nginx as a confused deputy to build a remote covert channel over the network. This covert channel reaches up to 8.9 bit/s. For both covert channels, we only use a single file for transmission, for simplicity, instead of the 256 and 512 for the process covert channel.

5.1 Covert Channel Across Virtual Machines

In this section, we show a covert channel between two virtual machines. We achieve up to 7.1 bit/s on the Samsung 990 EVO.

Threat Model. Two unprivileged processes are inside two virtual machines. They want to communicate because, e.g., one process wants to transmit a secret to the other one. The processes have no means of communication. Both processes have the permission to create files on their system. They also have a access to a shared clock. The VM disk is not cached on the host, as recommended, for example, by Red Hat [7]. To enable repeated tries to co-locate, the VM disk must also be configured with TRIM support or even better, with automatic trim when zeros are written [11]. We run our experiments on KVM with libvirt.



a. Step 1: Initial disk layout of sender 🟩 and receiver 🟧 files.

b. Step 2: After the sender accesses the whole file, the receiver gets all pages with cached timings: ▨.

c. Step 3: In the final step, the sender gets its pages with cached timings.

Fig. 5. The algorithm used to find all pages within the sender and receiver files that are in one HMB translation range. With five overlapping pages, five bits can be transmitted in one time slice.

Implementation. For this attack to work, we need HMB co-location between the VM disks for one VM to be able to influence the cache state of the other.

Co-Location. The communicating parties previously agree on a time when they start their covert communication. At the respective time, sender and receiver create a file containing random data. We assume that the operating system in the virtual machine, periodically uses TRIM to return unused disk space to the host, this is the default behavior, e.g., in Ubuntu. Therefore, the disk files of both virtual machines are sparse and grow with the created files, causing fragmentation and interleaving of the disk files.

We find to achieve maximum fragmentation and most overlaps between the two VM disk by writing in a three step process. In each VM, we write a 16 kB block, flush it to the disk using the 'sync' syscall and finally, punch a hole into this block, keeping only the first 4 kB. We repeat these three steps until we wrote between 50 MB and 200 MB. The steps are not synchronized between the VM.

Co-Location Detection. The virtual machine guest cannot get `FIEMAP` data of the host, therefore, it has to resort to another algorithm to detect which pages (4 kB blocks) of the files are within one HMB translation range, as shown in Figure 5. The sender repeatedly access a number of pages of their file to cache the translations. The receiver accesses their whole file and measures the access times. If the receiver measures a cached timing they store the offset. Then the receiver accesses the stored offsets that are potentially overlapping and the sender reads and measures the timings. After doing this for the whole file, the sender and receiver both know which offsets of their files are in the same HMB translation ranges. In a last step, the bit order must be measured. For this, the sender sends 1 bit every second and the receiver records the order in its file.

Transmission. We use the same time-sliced transmission as in the covert channel across processes, see Section 4.1. However, we only implement the transmission with a single bit per time slice for simplicity.

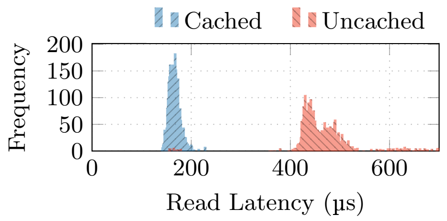


Fig. 6. Timing histogram measured from within a VM on SSD $\mathcal{C}$ (Samsung 990 EVO). It seems like libvirt multiplies the timing difference between the cache states, cf. Figure 3b.

Results. To measure the overhead introduced by the disk virtualization of libvirt, we transmit a toggling bit between two virtual machines and measure and plot the timings on the receiver side. Figure 6 shows the resulting histogram. The timings are clearly more separated than in the native scenario, cf. Figure 3b. On the Lexar NM790, the timings are even more separated, 569 µs ($n = 1000$, $\sigma = 267$ µs) vs 4 881 µs ($n = 1000$, $\sigma = 2\,298$ µs). We ruled out other reasons for this timing like the page cache. Our hypothesis is, that our highly fragmented disk files span many different HMB translation ranges and the read-ahead of the VM operating system, therefore, multiplies the uncached timing.

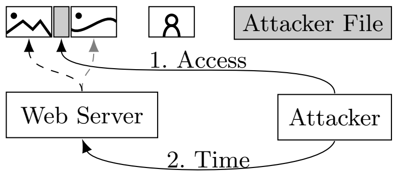
Co-Locations. For the covert channel to work, at least one fragment of the VM disk files each must be co-located in the same HMB translation range. The technique described above maximizes the chance of this happening. We run the file creation five times, always starting from a trimmed VM disk. On average we get 14091 ($n = 5$, $\sigma = 11002$) co-locations on the Lexar NM790. In one case no co-location was found. The larger HMB translation ranges of the two Samsung SSDs increase the number of co-locations respectively.

Transmission. We transmit random data to measure the transmission and error rate of the covert channel. Because we only transmit a single bit per time slice for simplicity, the transmission is considerably slower than with the process covert channel. However, the high average number of co-locations we found between the VMs would allow for multi bit permission with higher transmission rates.

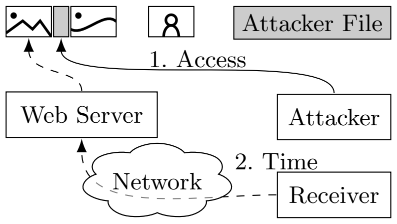
On the Lexar, the large delay for an uncached accessed and the resulting slow eviction time of 500 ms results in a slow transmission of 1.9 bit/s. However, due to the large separation between cached und uncached timings, we did not measure a single bit error over 400 transmitted bits. On the Samsung 990 EVO, we achieved a raw transmission rate of 7.1 bit/s, again without any errors over 400 transmitted bits. The HMB eviction takes 120 ms. On the Samsung 980 we, we achieved a raw transmission rate of 1.7 bit/s, again without any errors

5.2 Remote Covert Channel Attack

We show that an attacker can communicate through a benign website by inducing timing differences into HTTP responses. Schwarzl et al. [21] showed that timing differences of only 60 µs are measurable over 14 hops on the internet. This matches the timing differences on the Samsung SSDs.



a. Templating: The attacker accesses all offsets of the attacker file and learns about co-located assets by timing the web server.



b. While the attacker transmits a known sequence the receiver probes all assets and finds the one used for the transmission.

Fig. 7. The remote covert channel exploiting a benign web server as a confused deputy. The translation to one of the assets is either cached in the HMB or not.

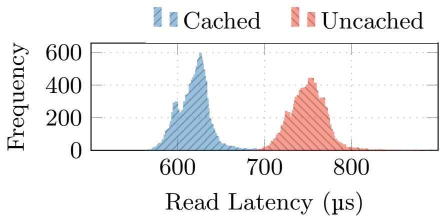


Fig. 8. Timing histogram of the Samsung 990 EVO measured over the network, exploiting nginx as a confused deputy. The distributions are spread out but overlap only barely, cf. Figure 3a.

Threat Model. We assume a threat model with two processes running on a system: First, a networked application, e.g., a web server like nginx, that is reachable by a remote attacker. Second, an unprivileged application that is isolated from the network but can access the web server through localhost, *i.e.*, the process has access to secret information but no outside network access. We assume that both application have access to a (non-overlapping) set of files on the disk, *i.e.*, they can perform disk operations but not directly communicate through any shared writeable or read-only files. For example, the attacker process has *no* access to the website assets served by the web server. This is typical, with the web server running with its own user. The web server uses direct disk I/O for large files, see Section 5.2. If the attacker had access to website assets, the templating phase could be skipped.

Communication Protocol. Figure 7 shows the attack overview. The basic idea is that the sender process either caches or not caches the translations to assets of the website. For this, the sender process must co-locate a file with a websites asset. If the receiver then accesses this asset they can infer if the translation was cached or not based on the response time of the HTTP request.

Implementation. For this attack to work we need HMB translation range co-location, direct I/O and low measurement overhead on the receiver.

Co-Location. To gain reliable co-location with many website assets, we create a large file, filling the whole disk for a brief moment. After creation of the file, we instantly free the first 90 % using `FALLOC_FL_PUNCH_HOLE`. The high disk pressure only persists for less than a second. In a final step, we punch more holes into the file to keep only one page per HMB translation range. Using the `FIEMAP` ioctl, we know which file offsets to free. This further reduces the size of our file, while we keep at least one page in many HMB translation ranges. If we do not yet have co-location with a usable website asset, there is a high chance that new assets will be co-located with fragments of our file.

Direct I/O. For this attack to work, we again have to evade the page cache. The nginx web server can be configured to use direct disk I/O with `O_DIRECT` on Linux when serving large files [16]. This has the advantage, that these big files then do not pollute the page cache. The nginx documentation recommends to enable it for files with 4 MB or more [16].

To minimize overhead and noise from the network transmission, we want to transmit as little data as possible. However, the asset served with `O_DIRECT` are multiple megabytes large. To circumvent this restriction, we use the range requests feature of HTTP [15]. It allows the client to request only a specified part of a file. We find that nginx always accesses the disk, when requesting at least 32 kB. Shorter ranges are still cached somewhere as we did not see nginx access the disk when requesting a shorter range more than once.

Templating. Before the transmission can start, the sender must find website assets co-located with its file fragments. First, the sender traverses the whole website to find files larger than 4 MB. Then it transmits a known pattern through up to 512 fragments of its file using the HMB and measures the access times to the assets though localhost. If the assets are large enough to span multiple HMB translation ranges, HTTP range requests are used to check multiple file offsets.

Sender. We again split one time slice in three parts. One for the sender to evict the HMB, the next to cache the required HMB translations, and the last for the receiver to measure the access time. We synchronize the sender and receiver using `clock_gettime` with the `CLOCK_REALTIME` clock source. After evicting the HMB, the sender accesses the co-located file fragment if the bit to transmit is a `1`, caching the translation in the HMB. While waiting, the sender periodically accesses the disk to prevent it from going into a low power state.

Receiver. The receiver measures the timing of the HTTP request, using the `Range` header to limit the response to 32 kB. While the receiver is waiting for the response to the HTTP request, Linux halts the core the receiver is running on, adding large timing variations to the timing measurement of the request. To mitigate this, we keep the core awake by running a busy loop in the sibling SMT thread while waiting for the response. At the beginning of the transmission, the receiver also has to traverse the whole website to get all large assets and then measure all of them to detect the one that is transmitting a known sequence.

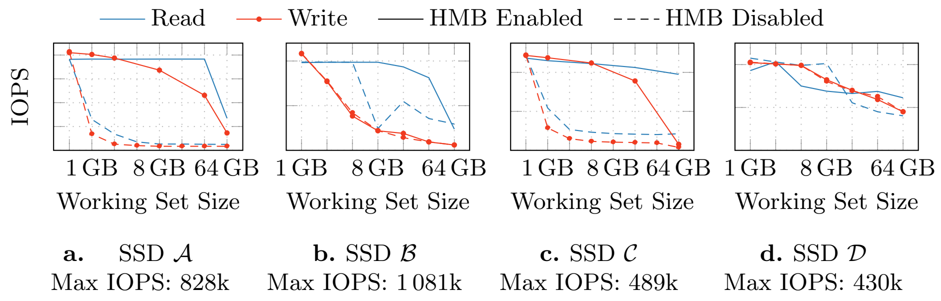


Fig. 9. The impact of disabling the HMB on random 4 kB IOPS at different working set sizes. The queue depth for the measurement is 32 with 16 parallel threads.

5.3 Results

We measured the covert channel using nginx/1.18.0 between two machines in the same network connected by one switch. Figure 8 shows the timing histogram of the Samsung 990 EVO. The added timing overhead from nginx and the network is not large enough for the two distributions to overlap, except for some outliers. This allows us to transmit 10 bit/s with an error rate of 1.5 %, resulting in a true capacity of 8.9 bit/s. The faster eviction on the Lexar NM790 allows for a raw transmission rate of 20 bit/s. However, the cached and uncached timing overlap, therefore, we use 5 measurements per bit, decreasing the raw transmission rate to 4 bit/s. With an error rate of 3.0 % this results in a true capacity of 3.2 bit/s. The slower eviction of the Samsung 980 slows down the covert channel to a raw capacity of 2.7 bit/s with an error rate of 0.9 %, for a true capacity of 2.5 bit/s.

6 Countermeasures

There are ways to mitigate the HMB side channel or make exploitation harder.

Disabling the HMB. Every SSD with the HMB feature must also work without host memory resources [17], albeit with reduced performance. On Linux the HMB can be disabled for all SSDs in the system with the kernel command-line parameter `nvme.max_host_mem_size_mb=0`, on Windows it is possible by setting the registry key `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\-StorPort\HMBAllocationPolicy` to 0.

Figure 9 shows the performance degradation when disabling the HMB. We measured the maximum achievable random IOPS when performing random 4 kB reads or writes with a queue depth of 32 and 16 threads on different working set, *i.e.*, file sizes. On SSDs $\mathcal{A}$ and $\mathcal{C}$, the IOPS of read and write accesses already decline by 80 % with a working set size of only 1 GB. They are practically unusable without the HMB. SSD $\mathcal{B}$ seemingly uses the HMB only for reads, they stay fast up until a working set size of 4 GB. We did not measure any difference in the write IOPS. On SSD $\mathcal{D}$ we measured a performance improvement when disabling the HMB over multiple measurements. However, it is also the SSDs least affected by the side channel. We did not see any differences in the sequential accesses on any of the SSDs. These results show, that depending on the workload and SSD, disabling the HMB can have a significant impact on the performance.

Modifying HMB Usage. We also see very different behavior of the HMB. The two newer PCIe 4.0 SSDs use LRU cache replacement which makes eviction very fast. On the Samsung 980 eviction takes more than 10 times longer and on the WD Blue SN550 we could not reliably evict the HMB. These older HMB implementations may have other disadvantages but they are superior security wise as they make attacks slower or impossible.

Making Software Interfaces Privileged. We use `O_DIRECT` to bypass the page cache for our timing measurements. Disabling it would make the attacks slower by multiple magnitudes, as the page cache would have to be evicted for every measurement. However, it is also used by benign applications like databases or as shown in Section 5.2 by nginx, for performance reasons. The usage of `O_DIRECT` could be controlled through a new capability that restricts its use to select programs. The same is true for the `FIEMAP` ioctl. Restricting untrusted applications from using it would greatly impede our attacks on accessible files.

7 Conclusion

In this work we show that the HMB feature of NVMe SSDs introduces exploitable timing differences measurable from use space. It is supported by almost all DRAM-less NVMe SSDs and we expect most of them to be exploitable. We evaluate four SSDs and reverse engineer how they use the HMB and analyze the timing differences and quick HMB eviction. Then we show four attacks exploiting the HMB. In the first two attacks we have access to the files we want to observe and build a covert channel with up to 8.3 kbit/s and a UI redress attack that detects the start of an application in less than 100 ms. In the blind attacks we use templating to find co-locations in the HMB and show a covert channel across VMs and network covert channel exploiting nginx as a confused deputy. Finally, we show that the HMB side channel can be mitigated by disabling the HMB, albeit with a significant performance impact on some workloads and SSDs.

Acknowledgements

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. Bernstein, D.J.: Cache-Timing Attacks on AES (2005), http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
2. Fratantonio, Y., Qian, C., Chung, S.P., Lee, W.: Cloak and dagger: from two permissions to complete control of the UI feedback loop. In: S&P (2017)
3. Gabriel, F.: SSD Database. https://www.techpowerup.com/ssd-specs/ (2025)
4. Giechaskiel, I., Tian, S., Szefer, J.: Cross-VM Covert- and Side-Channel Attacks in Cloud FPGAs. ACM Transactions on Reconfigurable Technology and Systems (2022)
5. Gras, B., Razavi, K., Bosman, E., Bos, H., Giuffrida, C.: ASLR on the Line: Practical Cache Attacks on the MMU. In: NDSS (2017)
6. Gruss, D., Kraft, E., Tiwari, T., Schwarz, M., Trachtenberg, A., Hennessey, J., Ionescu, A., Fogh, A.: Page Cache Attacks. In: CCS (2019)
7. Herrmann, J., Zimmerman, Y., Parker, D., Radvan, S.: Red Hat Enterprise Linux 7 - Virtualization Tuning and Optimization Guide (2019)
8. Juffinger, J.: An Analysis of HMB-based SSD Rowhammer. In: uASC (2025)
9. Juffinger, J., Rauscher, F., La Manna, G., Gruss, D.: Secret Spilling Drive: Leaking User Behavior through SSD Contention. In: NDSS (2025)
10. Kocher, P.: Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In: CRYPTO (1996)
11. libvirt: Domain XML format (2025)
12. Linux: Fiemap (2024), https://docs.kernel.org/filesystems/fiemap.html
13. Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-Level Cache Side-Channel Attacks are Practical. In: S&P (2015)
14. Liu, S., Kanniwadi, S., Schwarzl, M., Kogler, A., Gruss, D., Khan, S.: Side-Channel Attacks on Optane Persistent Memory. In: USENIX Security (2023)
15. Mozilla: HTTP range requests — MDN (2 2025), https://developer.mozilla.org/en-US/docs/Web/HTTP/Range_requests
16. nginx: directio — Docs ngx_http_core_module (2025), https://nginx.org/en/docs/-http/ngx_http_core_module.html#directio
17. NVM Express, Inc: NVM Express, rev 1.2.1 (2016)
18. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA (2006)
19. Pessl, P., Gruss, D., Maurice, C., Schwarz, M., Mangard, S.: DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security (2016)
20. Rydstedt, G., Gourdin, B., Bursztein, E., Boneh, D.: Framing attacks on smart phones and dumb routers: tap-jacking and geo-localization attacks. In: 4th USENIX Conference on Offensive Technologies (2010)
21. Schwarzl, M., Borrello, P., Saileshwar, G., Müller, H., Schwarz, M., Gruss, D.: Practical Timing Side Channel Attacks on Memory Compression. In: S&P (2023)
22. Schwarzl, M., Kraft, E., Gruss, D.: Layered Binary Templating. In: ACNS (2023)
23. Song, L., Pang, Z., Wang, W., Wang, Z., Wang, X., Chen, H., Song, W., Jin, Y., Meng, D., Hou, R.: The Early Bird Catches the Leak: Unveiling Timing Side Channels in LLM Serving Systems. arXiv (2024)
24. Trochatos, T., Etim, A., Szefer, J.: Covert-channels in FPGA-enabled SmartSSDs. ACM Transactions on Reconfigurable Technology and Systems (2023)
25. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security (2014)