# NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems

Zixuan Wang[⋆] Mohammadkazem Taram[♯⋆] Daniel Moghimi[†⋆]
Steven Swanson[⋆] Dean Tullsen[⋆] Jishen Zhao[⋆]

[⋆]UC San Diego [♯]Purdue University [†]UT Austin

## Abstract

We study microarchitectural side-channel attacks and defenses on non-volatile RAM (NVRAM) DIMMs. In this study, we first perform reverse-engineering of NVRAMs as implemented by the Intel Optane DIMM and reveal several of its previously undocumented microarchitectural details: on-DIMM cache structures (NVCache) and wear-leveling policies. Based on these findings, we first develop cross-core and cross-VM covert channels to establish the channel capacity of these shared hardware resources. Then, we devise NVCache-based side channels under the umbrella of NVLeak. We apply NVLeak to a series of attack case studies, including compromising the privacy of databases and key-value storage backed by NVRAM and spying on the execution path of code pages when NVRAM is used as a volatile runtime memory. Our results show that side-channel attacks exploiting NVRAM are practical and defeat previously-proposed defense that only focuses on on-chip hardware resources. To fill this gap in defense, we develop system-level mitigations based on cache partitioning to prevent side-channel leakage from NVCache.

## 1 Introduction

Microarchitectural side channels allow attackers to leak information from other users co-located on shared computing resources. Researchers have demonstrated such attacks by exploiting various hardware resources that are shared among untrusted users [16, 44, 48, 50, 54, 86]. For example, an attacker can construct a timing side channel based on the shared CPU cache and use this to break cryptography [6, 53], steal keystrokes [41], and violate the privacy of encrypted databases [64]. These side channels are also the basic block to developing more advanced microarchitectural attacks that leak arbitrary data and undermine the confidentiality of several isolation domains on modern systems [9, 35, 42, 58, 62, 68–70].

The research community has put a lot of effort into proposing defenses against microarchitectural attacks, but these proposals are not comprehensive. For attacks that target components internal to the CPU core [3, 16, 48], the recommendation is to isolate these resources for security-critical operations spatially and temporally [66, 67]. On existing CPUs with no fine-grain support for isolation, this can be achieved by flushing CPU resources across context switching or making sure untrusted threads are not simultaneously executed on the same core. For attacks on the shared CPU cache and its directory structure [44, 82], mitigations based on partitioning the cache and randomizing cache accesses are proposed [43, 56, 57, 78]. These CPU mitigations, promising for protecting security-sensitive computation, come with a performance penalty. Unfortunately, systems practically benefit little from paying the corresponding performance penalty to defend against CPU side channels if attackers can exploit other shared hardware resources such as the on-chip interconnect [33, 51, 63, 72, 74] and the DRAM banks and row buffers [55]. This gap motivates us to look into attacks on new memory subsystems.

We study the security implications of scalable server-grade non-volatile RAM (NVRAM) DIMMs as implemented by Intel's Optane DIMM [31]. NVRAM DIMMs enable a larger memory capacity and support for data persistence long desired by server developers. Recent performance characterization studies [76, 85] have shown that the Optane DIMM delivers its high levels of performance and scalability by employing various optimizations including multi-level buffers, internal address remapping schemes, and wear-leveling mechanisms. This combination leads to a discrepant performance behavior compared to what researchers expected before the product release [34, 37, 83]. Although previous studies have investigated the microarchitecture of the Optane DIMM and analyzed its performance, its security implications remain largely unexplored. In this study, we investigate microarchitectural covert/side channels enabled by Optane DIMM, their impact on the security of real-world applications, and how we can improve system security against potential side channels. More specifically, we contribute the following:

**1. Reverse-engineering (§ 3).** We perform reverse-engineering of the opaque design of NVRAMs, which helps

us uncover new information leakage sources. Our goal is to detail the on-DIMM cache structures and configurations, control policies, and performance behaviors. We develop carefully crafted microbenchmarks that run in both kernel and user spaces to achieve this. These microbenchmarks trigger specific memory behaviors, which lead to detectable performance variances that reveal the corresponding hardware designs. As a result, we unveil a much more detailed picture of Optane DIMM microarchitecture compared to previous works [76, 85]. Our findings include the on-DIMM cache structures and wear-leveling policies, which we then exploit to develop new information leakage attacks.

**2. Constructing covert channels (§ 4).** We develop and quantify new covert/side-channel attacks to empirically verify the existence of information leakage via the uncovered knowledge of Optane DIMM microarchitecture. First, we exploit the previously-undocumented on-DIMM cache structure to construct a cross-VM covert-channel attack. We show that cross-VM covert channels using the NVRAM cache are stable and achieve high channel capacity and low noise by solving several challenges. Second, we construct a covert channel that exploits the NVRAM wear-leveling mechanism to leak updates to a filesystem, which allows an attacker to monitor whether a victim updates its file without requiring elevated permission.

**3. Side-channel attack case studies (§ 5).** Next, we show that our findings go beyond covert communication channels and affect the security of real-world applications. We demonstrate several side-channel attacks exploiting the NVRAM cache, under the umbrella of NVLeak attacks, applicable to everyday use cases of NVRAM:

First, we demonstrate several attacks in the scenario where NVRAM is used as persistent storage, compromising the privacy of a SQL database or key-value storage. Although an attacker who shares the NVRAM with a victim does not have access to the victim's database/storage file and its queries, they can learn about its queries through NVRAM cache access patterns. Ultimately, an attacker can learn the details of queries and parameters, and previous work [19] shows that such information leakage is devastating for the privacy of encrypted databases.

Then, we demonstrate an NVLeak attack in the scenario where NVRAM is deployed like a volatile memory (like the DRAM). In this common scenario, to speed up workloads that don't require persistent storage, we show NVLeak can spy on code pages and detect which execution path is taken by a program whose code pages are stored in the NVRAM. Ultimately, we show that this has consequences for security-critical applications like cryptographic schemes.

**4. Mitigations (§ 6).** We propose a set of mitigation mechanisms to defend against the NVCache-based side channels based on the reverse engineering results and side-channel attacks. We first propose a software-based L2 NVCache mitigation that allows a victim application to allocate memory blocks from isolated NVCache sets that are not shared with other applications, including the attackers, thus preventing information leakages. We develop this mitigation into a PMDK [32] key-value store to make it resistant to NVCache-base side channels. The experimental results show that this mitigation's performance overhead is $< 4\%$. We then propose a software-based mitigation for L1 NVCache and WPQ, and a hardware-level mitigation for the entire NVRAM hierarchy.

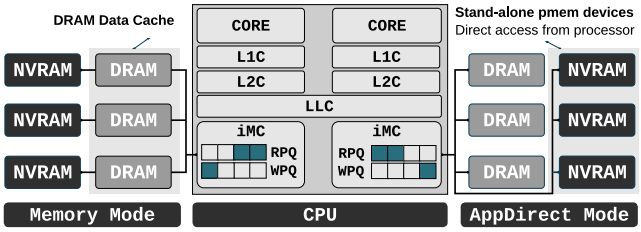



Figure 1: Memory hierarchy equipped with Optane DIMMs.

**Open source and responsible disclosure.** We open source our code[1] in the hope of facilitating future off-chip memory security research. We have disclosed the vulnerabilities and our code to Intel.

## 2 Background

In this section, we provide some background knowledge of the NVRAM and microarchitectural side-channel attacks.

### 2.1 NVRAM

Like DRAM, Non-Volatile RAM (NVRAM) provides a byte-addressable interface that allows the CPU to access NVRAM directly. NVRAM promises low-latency and high-bandwidth performance that is comparable to DRAM and faster than conventional NAND-based persistent storage devices [34]. On the other hand, NVRAM can also work as persistent storage, guaranteeing its data survives power reset, i.e., a user can rely on an NVRAM-aware filesystem for persistent storage [80]. Intel Optane DIMM [1] is the most widely adopted commercial NVRAM product.

**NVRAM-based server systems.** Figure 1 shows an example architecture of a server system equipped with Optane DIMM. Optane DIMMs (denoted as NVRAMs) are connected to the memory bus along with DRAM DIMMs. Intel's Cascade Lake processors, the first microarchitecture to support Optane DIMM [30], can support up to 6 Optane DIMMs per processor. The processor's integrated memory controllers (iMCs) manage access to the memory bus and connected memory modules, including the CPU's last level cache (LLC) and off-chip memory media. The iMCs have read-pending-queues (RPQs)

[1] https://github.com/TheNetAdmin/NVLeak

and write-pending-queues (WPQs) to process accesses to Optane DIMMs. WPQ, which is interesting for our work, stores the pending cache line data to be written back to NVRAM and ensures its data is safely persisted during power failures. Intel only documented the existence of WPQ [60] without further details such as WPQ size.

**NVRAM operation modes.** As shown in Figure 1, Optane DIMMs can be configured to operate in two different modes: Memory Mode or AppDirect Mode. In Memory Mode, a DRAM DIMM becomes a data cache for the Optane DIMM in the same channel. This mode allows users to enjoy the extra capacity provided by Optane DIMM, but it does not provide data persistence. In contrast, in AppDirect Mode, the Optane DIMMs are used as stand-alone persistent memory that is directly accessible to the software running on the CPU cores. The programmers can use regular load/store instructions to the NVRAM region and access in-NVRAM data structures. In this case, system software and NVRAM hardware should guarantee that the data structures are recoverable during system crashes or power outages [71, 80].

**NVRAM wear leveling.** Persistent storage devices such as SSDs use wear-leveling mechanisms to prolong the device service life: the persistent media's cells generally only support a limited number of write cycles [2], after which the cell is worn-out and not reliable to store data. The wear-leveling mechanism tries to evenly distribute write accesses to the media cells by migrating data from one cell to another after a number of write cycles. Optane DIMM uses 3D-Xpoint as its persistent media [47], which also requires wear-leveling to reduce the number of bad media cells. Currently, there is no publicly available documentation about the details of its wear-leveling algorithms.

**NVRAM microarchitecture.** Recent works [76, 85] unveil several previously unknown microarchitectural details of Optane DIMM. These works identify the existence of on-DIMM buffers organized as a hierarchy and the presence of a wear-leveling mechanism and a multi-DIMM interleaving scheme. In this work, we dive deeper into its microarchitecture, revealing more details and correcting some of the findings of the prior work. Thus, this paper represents the most comprehensive analysis of NVRAM to date, which also leads us to the first security analysis of an NVRAM microarchitecture.

## 2.2 Microarchitectural Side Channels

Microarchitectural side-channel attacks exploit contention on shared hardware resources such as shared caches [22, 23, 50, 52, 84], translation lookaside buffers (TLBs) [18], branch predictor structures [15, 16], DRAM buffers [55], and CPU execution ports [3, 8] to bypass software isolations and leak private information from other users. We discuss memory-related attacks in more detail due to the focus of our work:

**Cache attacks.** In a cache-based side-channel attack (cache attack), an attacker who cannot directly access the victim's data can observe the cache state changes made by the victim application. The attacker can monitor the state of the cache and derive the victim's secret information from its cache-access pattern. A typical cache attack comprises three main steps: (1) the attacker *prepares* the cache into a desired state so that the victim's cache activities are measurable. For example, the attacker prepares the cache by evicting specific cache lines from the shared cache. (2) the attacker *waits* for the victim to modify the cache state. (3) the attacker *measures* the cache state to determine whether the victim has accessed a target cache line or not. An attacker can achieve this by measuring the access time of its cache lines [52, 84] or the time of the victim's execution [50].

Cache attacks can be categorized into different classes based on their method for the preparation step. Flush-based attacks [21, 22, 84] assume a shared memory (e.g., a shared library) between the attacker and the victim, so the attacker can prepare the cache by simply executing a `clflush` [26] instruction to evict target cache lines. In contrast, conflict-based attacks (e.g., Prime+Probe [52]) do not rely on shared memory. Instead, to prepare the cache states, the attacker identifies and loads multiple cache lines mapped to the same cache set as the target address, forcing the cache to evict the victim's target cache line due to conflicts.

**DRAM row-buffer conflicts.** Previous research has also attacked on-DIMM structures. These attacks leak secrets by exploiting DRAM-specific optimizations and features such as row buffers [55] and row activation [36]. DRAMA [55], for example, exploits the DRAM row buffer, an on-DIMM SRAM memory that caches an entire DRAM row and provides faster access than DRAM banks. DRAMA performs a Prime+Probe attack by leveraging the timing difference between row buffer hits and misses to detect victim activities such as keystrokes.

# 3 NVLeak Reverse-Engineering

In this section, we perform reverse-engineering of the NVRAM's microarchitecture. We build our experiments on top of LENS [76], a recent performance characterization study of Optane DIMM. Our results confirm some of LENS' findings, including the capacity of DIMM caches and their block size on two different versions of the Optane DIMM. Additionally, we recover undocumented details such as the set associativity of the DIMM caches and their indexing schemes (§ 3.2). Finally, we correct some essential details about the wear-leveling policy and its implementation (§ 3.3).

## 3.1 NVRAM System Configuration

Table 1 shows the configuration of the NVRAM-equipped machines we use for our experiments. We configure all DIMMs to be non-interleaved to ensure that our experiments always

Table 1: NVRAM-equipped server system configuration.

| | Server A | Server B |
|---|---|---|
| CPU | Intel Xeon Cascade Lake<br>24 Cores per socket, 2 sockets<br>HyperThreading off | Intel Xeon Gold 6230<br>20 Cores per socket, 2 sockets |
| L1 Cache<br>L2 Cache<br>L3 Cache | 32 KiB 8-way I-Cache, 32 KiB 8-way D-Cache, private<br>1 MiB, 16-way, private<br>33 MiB, 11-way, shared | 27.5 MiB, 11-way, shared |
| DRAM | 6 channels per socket<br>DDR4, 32 GiB, 2666MHz | DDR4, 16 GiB, 2666MHz |
| NVRAM | Intel Optane DIMM<br>6 channels per socket<br>256 GiB, 2666 MHz<br>Firmware: 01.01.00.5253 | 128 GiB, 2666 MHz<br>Firmware: 01.02.00.5355 |
| Kernel | Linux 5.4.0 | |

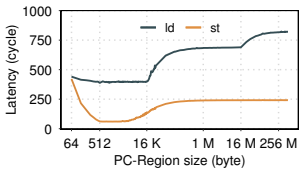
run on a single DIMM. Additionally, we configure Optane DIMMs to operate in AppDirect mode. For our reverse-engineering microbenchmarks, we mount a dummy filesystem on an Optane DIMM which provides identity memory mapping on the Optane DIMM memory region. To minimize the noise in our experiments, we turn off the CPU hardware prefetchers, disable simultaneous multithreading (SMT), and boost the CPU to performance mode with the help of the CPU scaling governor. This section presents results using Server A (Table 1), and § A.2 shows results on Server B.

## 3.2 Recovering Details of NVCache and WPQ

We design a pointer chasing microbenchmark that reveals the structure of the NVRAM cache (NVCache) and CPU's write-pending-queue (WPQ). Here, we first describe the microbenchmark, then briefly discuss how we instantiate this program to confirm the results of LENS on cache capacity, block size, and WPQ structure. We then discuss how we recover NVCache set associativity, the number of sets, and NVCache indexing scheme.

**Pointer chasing microbenchmark.** We divide a contiguous memory region—a pointer chasing region (PC-Region)—into equal-sized blocks (PC-Blocks). We initialize each PC-Block with a pointer to another random PC-Block. Recursively dereferencing the pointers results in accessing all PC-Blocks in random order. This also defeats a cache prefetcher if that exists. To minimize the effects of CPU caches on our measurements, we set the NVRAM memory region as *uncacheable* through x86 Memory Type Range Register (MTRR) [26]. We ensure that every read or write operation reaches the NVRAM DIMM by using non-temporal load and store instructions. In an environment where MTRR is not available, e.g., user-space processes or virtual machines, one can instead insert a `clflush` instruction after each memory access instruction to imitate the uncacheable effects.

**Recovering cache capacity.** To identify the cache capacity, we follow the LENS [76] methodology and run the




(a) Pointer Chasing

| PC-Block | L1 NVC | L2 NVC | WPQ |
|---|---|---|---|
| 64B | 1.73 | 1.21 | 6.65 |
| 128B | 1.43 | 1.17 | 3.80 |
| 256B | 1.00 | 1.17 | 1.85 |
| 512B | 1.00 | 1.12 | 1.00 |
| 1K | 1.00 | 1.07 | 1.00 |
| 2K | 1.00 | 1.03 | 1.00 |
| 4K | 1.00 | 1.00 | 1.00 |

(b) Amplification

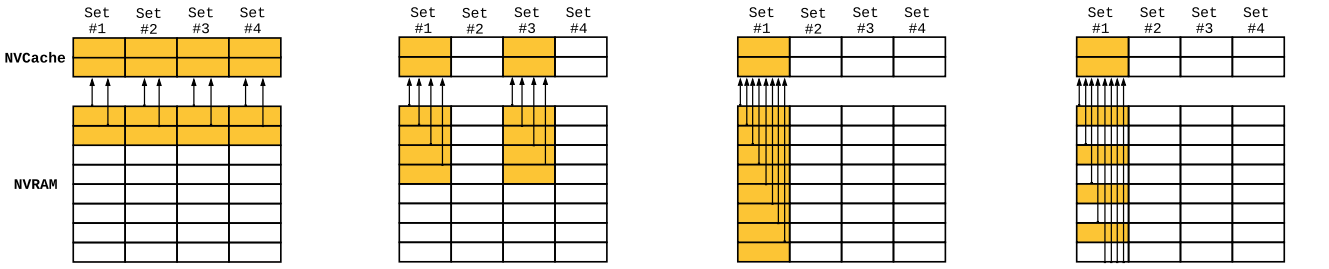
Figure 2: Pointer chasing microbenchmark results and amplification factors for each architectural components.

pointer chasing microbenchmark with varying region sizes (PC-Region) while keeping the PC-Block size constant. The results of this experiment (Figure 2a) confirm the presence of two levels of NVCaches with 16 KiB and 16 MiB capacities on both of our machines. Hereafter, we refer to the 16 KiB cache as L1 NVCache and the 16 MiB one as L2 NVCache.

**Recovering L1/L2 NVCache block size.** To confirm the block size reported by LENS [76], we use the pointer chasing microbenchmark with varying PC-Block sizes and measure the read/write *amplification factor*. The read/write amplification factor is the ratio of data that is read/written to the size of the requested data. For example, for a cache with 256 B block size, a single read request of 64 B will bring a 256 B block into the cache; thus, we will have a read amplification of 4. Our read/write amplification results (details in Figure 2b) confirm that the two NVCaches have different block sizes. L1 has 256 B blocks while the L2 NVCache has 4 KiB blocks.

**Recovering CPU WPQ structure.** Figure 2a shows that pointer chasing write latency drops until 512 B region size, which indicates a buffer or queue on the write path. Given its small size and presence only on the write path, we believe it is the write pending queue (WPQ) embedded in the Intel CPU's on-chip memory controller [60]. The WPQ's write amplification factor (Figure 2b) drops to 1 when using 512 B PC-Block; considering the pointer chasing microbenchmark inserts one `mfence` instruction after each PC-Block access, we believe that one `mfence` instruction flushes the entire WPQ.

**Recovering NVCache associativity and number of sets.** We run the pointer chasing microbenchmark with strides to identify the number of cache sets, which prior works [76, 85] have not revealed. As illustrated by Figure 3, increasing the stride size directs the pointer chasing accesses to fewer cache sets. For example, assuming a target cache has four sets (unknown to the attacker) where each set stores two cache blocks, and the cache uses a linear indexing function to index the sets. If the attacker runs the pointer chasing code to access eight continuous cache blocks, the attacker can fill up the entire cache (Figure 3a). But if the attacker doubles the stride size (Figure 3b), its accesses will skip every other set, and the attacker can only use half of the cache sets. Finally, once the attacker reaches a stride that matches the number of sets, all memory accesses will be mapped to only one cache set




(a) Stride Size = Cache Block Size (b) Stride Size = 2 * Cache Block Size (c) Stride Size = 4 * Cache Block Size (d) Stride Size = 8 * Cache Block Size

Figure 3: Detecting NVCache set structures with a strided pointer chasing microbenchmark. In this example, NVCache has 4 sets, where each set has 2 cache blocks. The strided access reaches a maximum cache contention once the stride size reaches 4 times the cache block size. Figure (d) is cut off with 4 more memory block accesses out of the figure range.

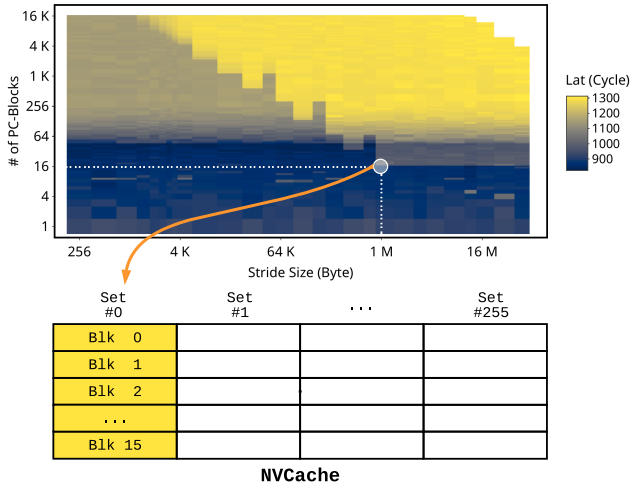



Figure 4: Pointer chasing microbenchmark results. The heatmap shows the pointer chasing latency under varying striding sizes and number of blocks. When we have 16 blocks or less, the latency does not increase for any stride size, indicating that we have at least 16 block available, i.e., associativity of 16.

(Figure 3c). At this point, doubling the stride size does not reduce the available cache capacity anymore (Figure 3d).

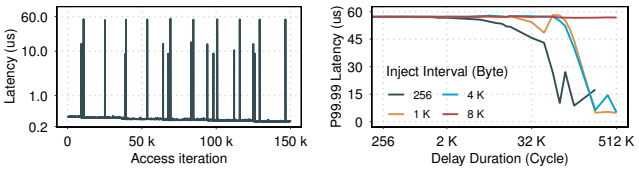
Using this method, we scan all possible stride sizes and measure the available cache capacity, and find a point where the cache capacity stops decreasing. We can then use that stride size to deduce the number of sets in the cache, and since we already know the total number of blocks in the cache, we can also derive its set associativity.

Figure 4 shows a heatmap of the pointer chasing read access with different stride sizes and different numbers of blocks. We repeat the pointer chasing with strides for multiple rounds (16 or more) and measure the average latency to get a stable result. We use a 64 B pointer chasing block size[2] so that each NVCache block gets only one memory access.

*L1 NVCache set structure.* We start with a stride size of 256 bytes. This stride size matches the L1 NVCache block size, so all pointer chasing accesses reside in different consecutive L1 blocks. We observe a rapid sizable increase in the read latency at 44 blocks. This overflow point is close to L1 total blocks, so we believe this reflects the L1 NVCache overflow. This overflow point remains at 44 blocks under various stride sizes, which means that the mapping of blocks to cache sets does not depend on their addresses, indicating a fully-associative structure. These results also suggest that L1 employs a replacement policy different from Least Recently Used (LRU). An LRU cache should overflow when it is full, but L1 starts to overflow at 44 instead of 64 blocks (the maximum capacity of the L1 NVCache).

*L2 NVCache set structure.* When stride size is set as 4 KiB, which matches the L2 block size, every memory access falls into a different L2 block, and we observe an additional distinguishable latency change at 4096 blocks. The first overflow point at 44 blocks reflects the L1 NVCache capacity, while the second one matches the size of L2 NVCache (4096 blocks $\times$ 4 KiB = 16 MiB = Size(L2 NVCache)). Unlike L1, the L2 overflow point changes with the pointer chasing stride size: it overflows at 4096 blocks with 4 KiB stride size, then at 2048 with 8 KiB stride, etc. The overflow point stops at 16 blocks with 1 MiB stride and does not change with larger stride sizes. This indicates that L2 is a set-associative cache because its overflow point changes and each L2 set has 16 cache blocks because the overflow point stops moving at 16 blocks. This result also shows L2 likely uses LRU replacement, as L2 starts to overflow with 4096 blocks which match the number of L2 blocks, i.e., L2 overflows when it is full. L2 NVCache is likely to use a linear indexing scheme because there is a linear relationship between the number of PC-Blocks and the stride size for L2 overflow points.

[2] 64 B is the AVX512 register size we use and is unrelated to the NVCache parameters. SSE and AVX2 registers can also be used with non-temporal load/store instructions.




(a) Per-iter latency. (b) 99.99 percentile latencies.

Figure 5: Overwrite microbenchmark results. (b) shows a sample of results from Figure 6.

## 3.3 Recovering Wear-Leveling Policy

Next, we recover the wear-leveling policy used in the Optane DIMM. To that end, we extend the *overwrite* microbenchmark from LENS [76] to measure the timing of the data migration and its spatial granularity (i.e., the block size).

**Triggering a data migration.** We start with a variant of the overwrite microbenchmark that repeatedly writes 256 B regions and measures the latency of each write operation. If a write in this region triggers a data migration, e.g., by a wear-leveling algorithm, the subsequent writes cannot be issued until the migration completes. As a result, a write stalled by a migration exhibits a latency over a magnitude higher than an average write. We estimate the migration latency using the elevated tail latency. Furthermore, we can calculate the migration frequency by measuring the time intervals between two consecutive migrations.

Figure 5a shows the results of the overwrite microbenchmark for data migration latency analysis. The results of Figure 5a show that long-latency writes (∼60 us) happen at regular intervals. We see one after writing every ∼4 MiB (∼16 K write operations) to the memory region when using non-temporal stores. This confirms the behavior observed by previous work [76]. We also performed the same overwrite experiment on DRAM and did not observe such a long latency effect. Thus we consider the cause of this NVRAM long write latency is its data migration for wear-leveling or thermal control.

**Data migration timing conditions.** To further investigate the conditions upon which a data migration happens, which is not covered by prior works [76, 85], we examine the effects of write frequency in triggering long delays. We inject delays with varying lengths in the overwrite microbenchmark and measure the 99.99 percentile latency. We can inject delays via two knobs: the delay duration, which indicates the number of CPU cycles that we wait during each delay, and the delay injection interval, which determines how much data we write between two consecutive injected delays. By changing the delay duration and injection interval and observing the tail latency, we can deduce the timing conditions for data migrations, i.e., the correlation between the data write frequency and the data migration frequency.

Figure 6 shows the results of this experiment. Each point

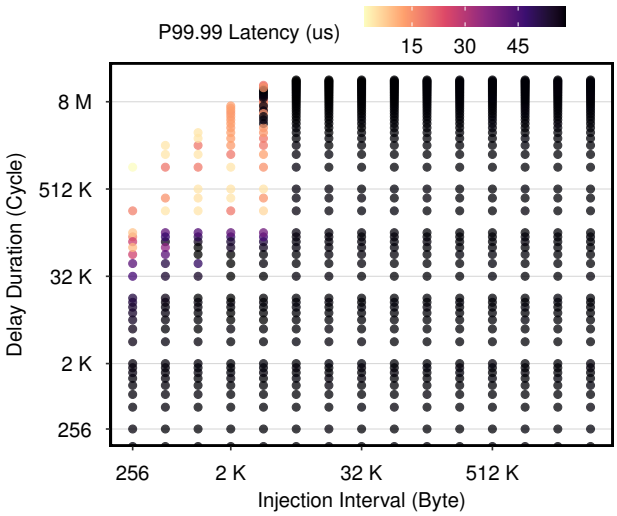



Figure 6: Overwrite 99.99 percentile (P99.99) latencies under various delay duration and delay injection interval settings.

represents a 99.99 percentile latency of the overwrite microbenchmark that writes 512 MiB data to the same 256 B NVRAM region. We observe that injecting longer delays reduces the frequency of long latency writes significantly. This might be due to the data migration process hidden in the delays. So, when the delay duration is too long, the migration process is likely to happen when we execute delays, and thus none of our writes encounters a long latency. In addition, we observe that overwrites trigger the maximum amount of long latencies (i.e., highest P99.99 latency) if the delay is infrequent (injection interval $\geq$ 8 KiB) or delay duration is short (duration $<$ 32 K cycle).

**Data migration spatial granularity.** We run two threads writing data to NVRAM simultaneously but on different regions to detect the migration granularity, i.e., the block size for each migration. Two threads inject delays at different frequencies: one *scratch* thread injects no delays, and the other *probe* thread injects delays that do not trigger the full amount of long latency (the migration timing conditions identify this delay setting). When two threads are working within the same migration block, the probe thread observes the full amount of long latencies due to the scratch thread's activity. Once two threads' regions are distant enough to guarantee they do not share a migration block, the probe thread is the only writer working on its region, thus observing few long latencies.

On our machine, we set the probe thread to inject a 128 K cycle of delay after each 256 B write, gradually increase the delta between the two threads' starting addresses, and observe that the probe thread's P99.99 latency drastically drops when the delta reaches 4 KiB or larger. This indicates that two threads are now separated into two different data migration blocks. Thus, the migration block size is 4 KiB.

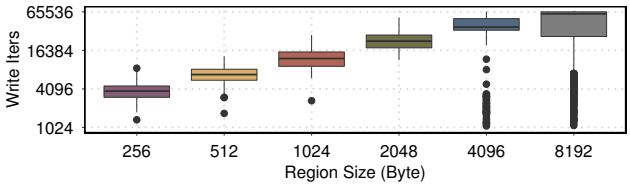



Figure 7: Data migration robustness when using various memory regions. The y-axis shows the number of 256 B writes it requires to trigger one data migration. Each bar shows the distribution of write iterations.

**Data migration robustness.** Next, we examine the distribution of the number of writes needed to trigger a single data migration. We repeatedly write to a memory region, where each repeat consists of sequential writes to each 256 B block within the region. We then count how many writes it requires to trigger one data migration (latency over 100 K CPU cycle). In this experiment, we use the `clwb` instruction instead of the non-temporal stores because `clwb` executes faster and generates more frequent writes to NVRAM, which trigger more frequent data migrations and give us a more accurate observation of the data migration robustness.

Figure 7 is a box plot that shows the results of the above-mentioned experiment. The figure shows that it is easier to trigger a migration when overwriting a smaller region, and at minimum, it takes ∼3,700 iterations (i.e., 3,700 × 256 = 925 KiB) to trigger one data migration. This also shows that for a region size of 256 B, 50% of the time the required number of writes is within the range of `[3027:4552]`. The migration happens at 4 KiB granularity, i.e., one migration affects the entire 4 KiB page, but any of its 256 B blocks can trigger it. We also observe a more frequent data migration in Figure 7 (every ∼4 K writes) compared to Figure 5a (every ∼16 K writes) due to `clwb`'s faster execution than non-temporal stores.

**Feasibility of side channel.** We analyze the feasibility of establishing a side channel based on this data migration mechanism, using a proof-of-concept code: the victim and the attacker shares the write access to a 4 KiB page and update different regions. The victim writes to its region at the highest frequency, while the attacker updates the region at a frequency that does not trigger data migration on its own. In this POC, the attacker observes frequent data migrations when the victim is updating the shared page, compared to when the victim is idle. We evaluated this POC and found that the attacker can detect if the victim has written more than one MiB of data to the shared page, which aligns with the data migration robustness analysis (it takes ∼925 KiB writes to trigger one migration). This channel may leak the information when the victim application repeatedly updates metadata or a file: e.g., an attacker times a database row update to detect if there are updates to nearby rows by the victim.

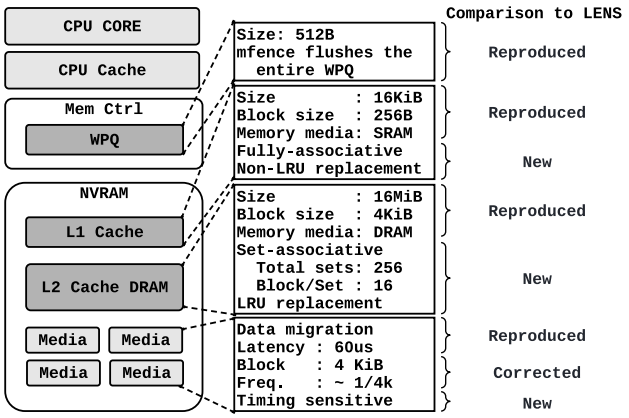



Figure 8: Overview of the NVRAM findings. *Reproduced* results are from LENS, *corrected* results are from NVLeak and correct observations from LENS, and *new* results are from NVLeak and not discovered by LENS.

## 3.4 Summary of Findings

Figure 8 shows an overview of our findings:

- Optane DIMM has two data caches (NVCaches), organized as two-level inclusive caches: The Level 1 NVCache is a fully-associative cache with 16 KiB size and 256 B block size. The Level 2 NVCache is a 16-way set-associative cache with 16 MiB size and 4 KiB block size.
- The size of the write-pending-queue [60] is 512 B.
- There is a long-latency effect (60 us) when frequently flushing data to NVRAM.
- The data migration happens after ∼4 K writes to the same migration block of 4 KiB.

Compared to LENS [76], we have corrected a set of observations, and discovered previously unknown architecture designs, as shown in Figure 8.

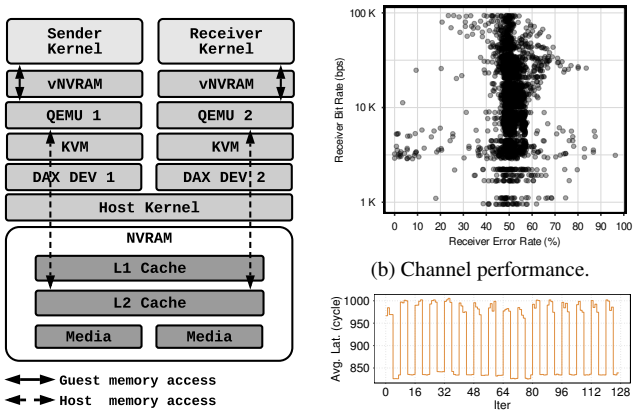
# 4 NVLeak Covert Channels

Based on our findings from the previous section, we demonstrate covert channels across isolated security domains. We first demonstrate a cross-VM covert-channel attack based on the Prime+Probe attack technique [52] exploiting the on-DIMM NVCaches. We then demonstrate a wear-leveling-based covert-channel attack across filesystem boundaries. These attacks are demonstrated on Server A (Table 1).

## 4.1 Cross-VM NVCache Channel

This section presents our cross-VM covert-channel attack exploiting the level 2 NVCache.

**Threat model.** We assume the sender is a program running inside a guest VM that maps an NVRAM memory region




(a) Overview.

(b) Channel performance.

(c) Channel receiver signal.

Figure 9: Cross-VM covert channel. (c) is a receiver signal from (b) that achieves 11 kbps bandwidth with 3.5% error rate. This channel uses 14 blocks, 1 MiB stride size and 16 repeat rounds. Y-axis shows the average per-block latency.

from the host system (the virtualization hypervisor) to the guest VM. This region appears as a virtual NVRAM device in the guest VM. We assume the receiver is another guest VM that maps a different NVRAM memory region. We assume the sender does not have any shared memory with the receiver; their NVRAM memory regions are not overlapping and can belong to two different partitions, as long as these regions are both on the same NVRAM DIMM. We assume the sender and receiver have full control of their own guest VM's software, i.e., they can replace OS kernels or run code at the kernel level within their guests.

**Challenges.** To obtain a stable, low-noise, and high-resolution channel, we overcome the following challenges:

*Challenge 1: Bypassing L1 NVCache.* Our reverse-engineering results (§ 3) showed that the L1 NVCache is a fully-associative on-NVRAM cache shared by all cores. Its fully-associative property introduces noise and reduces the resolution of the channel. To bypass the L1 NVCache effect in our Prime+Probe attack, we must flush the L1 NVCache before every Prime step. To achieve this, we read data from an NVRAM region different from the Prime memory region. This guarantees the Prime step's memory accesses are cache misses in the L1, thus bypassing the L1 and directly accessing L2.

*Challenge 2: Targeting L2 NVCache sets.* We need to Prime and Probe each L2 NVCache set separately to construct a high-resolution channel. We rely on the knowledge of L2's indexing scheme we discovered in § 3—L2 takes bits [13:20] of each address—and use it as an index to a set. Thus, to evict a specific L2 set, we first identify the set address of the target L2 set, then create our conflict set with a set of 16 blocks that are based on the set address and are 1 MiB apart. This ensures that all of these blocks go to the desired set.

*Challenge 3: Avoiding noises due to cache prefetchers.* There are multiple prefetchers for on-CPU caches and on-NVRAM caches in our system. These prefetchers cause latency fluctuations in the Prime+Probe accesses, adding noises to the channel. To defeat these prefetchers, we use our pointer chasing access pattern, which accesses all memory blocks, but in a random order so that prefetchers cannot detect a predictable pattern.

*Challenge 4: Synchronizing the parties.* Without synchronization, the sender and receiver can easily drift, making the received bit sequence unreliable, especially for long bit sequences. To overcome this challenge, we implement a timing-based synchronization, where the sender and receiver use the same predefined time interval for each bit send/receive iteration: After the pointer-chasing operation finishes, the thread waits until the time interval is finished, then starts the next iteration. From our evaluation, this timing-based sync introduces ∼10% additional waiting time compared to the pointer-chasing operation time.

**Evaluation.** We configure a single-DIMM NVRAM into two non-overlapping DAX devices (Figure 9a). Each DAX device is passed to KVM and QEMU to create a virtual NVRAM for the guest VM.

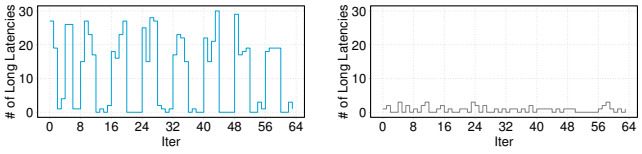
We use the KVM-unit-tests framework [40] to implement two OS kernels for the sender and receiver to run at ring-0 within the guest VM. KVM-unit-tests provide a set of libraries to build customized OS kernels, and each of its test cases is a small kernel that boots up from a guest VM and runs a piece of testing code. We implement the sender and receiver as two test cases in KVM-unit-tests that run covert channel code at kernel level or ring-0 level.

Figure 9b shows the performance of the cross-VM covert channel. Each point in the figure represents one covert-channel experiment that sends/receives a binary data sequence. Different points use different configurations for the underlying pointer-chasing microbenchmark, including region size, block size, stride size, repeat rounds, and flushing L1 NVCache or not. From Figure 9b we find the best configuration achieves 11 kbps bandwidth with 3.5% error rate (Figure 9c).

## 4.2 Data-Migration-Based Channel

This section describes how we construct a covert channel that exploits the NVRAM-specific wear-leveling data migration mechanism.

**NVRAM data migration.** As described in § 3, the Optane DIMM uses a wear-leveling data migration mechanism that triggers long latencies in response to frequent writes to the same migration block. This migration mechanism is sensitive to the memory write frequency: the migration happens less frequently (relative to the write traffic) once the write frequency drops to a certain point (e.g., one access per 32 K cycles as shown in Figure 6).




(a) NVRAM (b) DRAM

Figure 10: Filesystem inode-based covert channel on NVRAM and DRAM.

We exploit the migration latency to construct a covert-channel attack where the receiver and the sender write to the same NVRAM data migration block. The receiver uses a low-frequency write and measures write latencies. If the sender frequently writes to the same migration block, the block gets migrated more regularly, which results in long-latency writes detected by the receiver.

This attack requires the receiver and the sender to write the same 4 KiB migration block. However, this write access does not need to be direct. For example, we can assume a receiver and a sender own two separate files, but the files' metadata resides in the same migration block. In such cases, any write to the sender's file results in a metadata update, and the metadata update involves a write to the migration block. Neither thread has direct access to the metadata, but they leak information through underlying filesystem operations.

**Threat model.** We assume the receiver is a user-space program with access to a file. This file's *inode*–Linux filesystems metadata—is stored in a metadata page that also stores the sender's file inode. The filesystem maintains the inode, and the user space program cannot directly modify a file's inode. However, a user space program can use (1) `futimes()` to update a file's access and modify time, or (2) `ftruncate()` to update the file size, without the overhead of an actual file data update. The size of an inode is typically sub-page-size, e.g., inodes in the EXT4 filesystem are 256 bytes [39]. Thus one metadata page (4 KiB) stores multiple inodes, enabling the receiver to access a file that shares a metadata page with the sender's file.

**Cross-filesystem covert channel.** We construct a covert channel based on the filesystem's inode updates triggered by the data migrations. The sender program updates its file's metadata at maximum frequency if sending bit 0; otherwise, it remains idle for a predetermined period to communicate bit 1. The receiver constantly updates its file at a low frequency to not trigger data migrations on its own but to detect that of the sender. The receiver also times these inode update functions and identifies the data migration as a long latency. The receiver picks up a bit 0 if it detects data migrations.

The widely-adopted EXT4 filesystem allocates inodes sequentially, i.e., for each newly created file, EXT4 allocates the first available inode from a list of free inodes sorted by the inode number. The receiver can exploit this allocation mechanism to create its file at roughly the same time the sender creates its files so that their inodes are placed on the same metadata page with a high probability.

```
-- U1: Update 100 records in 'info' table
UPDATE info SET name = "New Name" WHERE npi == 1144223363;
-- U2: Update 1 record in 'address' table
UPDATE address SET city = "New City" WHERE npi == 1144223363;
-- C1: Count records in 'address' table
SELECT COUNT(*) FROM address WHERE LOWER(city)='athens';
-- C2: Count records in 'info' and 'address' tables
SELECT COUNT(*) FROM info, address WHERE
  info.npi = address.npi AND LOWER(city)='athens';
-- C3: Count records in 'info' and 'address' tables
SELECT COUNT(*) FROM info, address WHERE
  info.npi = address.npi AND LOWER(city)='houston';
-- Q1: Query both tables
SELECT * FROM info, address WHERE
  info.npi = address.npi AND info.npi == 1144223363;
-- I1: Insert 10,000 records in 'info' table
INSERT INTO info values (...), (...);
-- S1: Sort records in 'address' table
SELECT * FROM address WHERE LOWER(city)='athens'
  ORDER BY city DESC LIMIT 1;
```

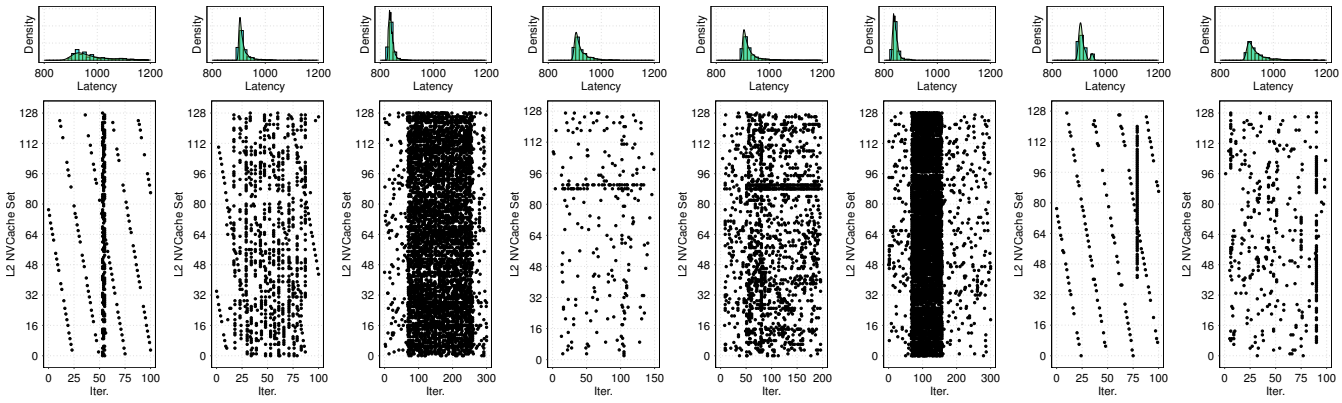
Figure 11: Evaluate SQL operations.

**Evaluation.** In our evaluation, we configure the NVRAM into filesystem DAX mode and create an EXT4 filesystem on the NVRAM. We mount the EXT4 filesystem in DAX mode, which disables the filesystem page cache and allows direct access to the underlying NVRAM. This is the preferred use case of a filesystem for NVRAM [38].

Figure 10a shows the receiver signal on an NVRAM-based inode channel, where the sender sends `0xcc` followed by repeated `0xf0`. In this example, this channel achieves 217 bps bandwidth with 0% error rate. The bandwidth can be further improved to 1.5 kbps if we reduce the number of inode updates for each bit. For comparison, we also evaluate the possibility of covert communication if we use DRAM instead of NVRAM. As shown in Figure 10b, the DRAM does not achieve a stable channel as it does not implement the data migration for wear-leveling.

## 5 NVLeak Side-Channel Attacks

We demonstrate three types of attacks on Server A (Table 1) where victims access NVRAM through different interfaces: (1) attacking a database which accesses NVRAM through conventional filesystem interface, where the attacker exploits L2 NVCache to learn information about SQL queries (§ 5.1); (2) attacking an NVRAM-optimized key-value store which accesses NVRAM through NVRAM-aware filesystem and libraries, where the attacker learns the in-flight key-value pair information (§ 5.2); (3) attacking a victim program that links with shared libraries that are stored in NVRAM and accessed without page cache, where the attacker exploits L1 NVCache to learn the victim code execution path (§ 5.3).




(a) U1 (b) U2 (c) C1 (d) C2 (e) C3 (f) Q1 (g) I1 (h) S1

Figure 12: Access patterns of database operations. (a)-(h) each shows the access pattern for a database operation: the bottom figure shows the latencies in a concentrated range of iterations and sets, and the top figure shows the latency distributions.

## 5.1 Database Operation Leakage

This section describes an NVCache-based side-channel attack that can violate the privacy of encrypted database applications.

**Threat model.** Motivated by previous work on cache-based side-channel attacks on encrypted databases [64], we assume the victim is a database application that accesses data stored in NVRAM, and the attacker runs in another process or virtual machine. We assume an attacker who does not have direct access to the database and whose goal is to learn about the queries that the victim runs. The attacker repeatedly probes all the NVCache sets to detect the victim's NVRAM access pattern. The attacker first records access patterns of different database SQL operations and uses them to categorize newly detected patterns.

**System and software configuration.** We configure an Optane DIMM into two partitions, one for the victim database to store files, and the other one for the attacker to probe the NVCache. We use sqlite3 [65] as the victim database, and create the database tables from the NPPES dataset [11], which results in a 286 MiB database file. There is an *info* table with users' basic info (e.g., full names) and National Provider Identifiers (NPI) as primary keys. There is another *address* table that stores user addresses, including city and state, and NPI as a foreign key referencing the info table.

**Leak SQL statements.** Figure 12a to Figure 12g show the access pattern under different victim database operations (Figure 11), detected by the attacker program. U1 and U2 are updating to two different tables, and their access patterns are visually different: U1 only affects 1-2 attacker iterations, while U2 affects 70 iterations. U1 and U2 also have different latency distributions, as shown on top of the access pattern figure. U2's latencies are more concentrated around 900 cycles, while U1's are more evenly distributed. All other operations have different access patterns and latency distributions.

We use the k-nearest neighbors algorithm [4] (k-NN) with Pearson correlation coefficients to analyze and categorize these access patterns quantitatively: We first identify a latency threshold range to filter out the non-relevant latencies and then encode the latencies into a binary array, where bit 1 represents the corresponding latency within the threshold. We use this binary array as the *feature* of the corresponding database operation.

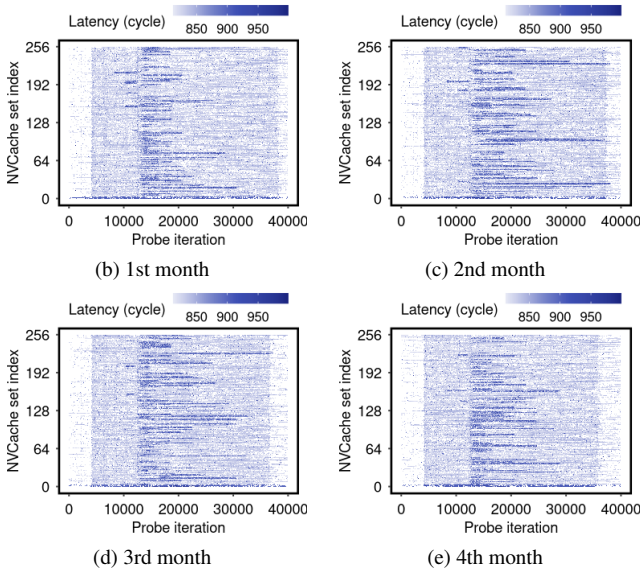
To categorize a newly detected database operation, we calculate the Pearson correlation coefficients between its feature and previously-recorded features, and use their coefficients as *distances* in the k-NN algorithm. We categorize this newly detected operation as one of the previously-known operations that takes the majority of the k nearest neighbors of the new operation. To evaluate this categorization algorithm, we run each SQL operation 100 times, then for each operation, randomly choose 70 samples as the k-NN training set and use the rest 30 samples as the test set. In this evaluation, 240 SQL operations are tested and 202 of them are correctly categorized, resulting in an accuracy of 84%.

**Leak SQL range query.** We then demonstrate that the NVCache-based side channel can leak more detailed information in database accesses. As demonstrated by previous work [19, 64], leaking range query details is specifically critical for encrypted databases where learning about these queries can almost result in a complete loss of privacy. As an example, we assume the victim executes a range query (Figure 13a) based on the creation date of NPPES data records; And the attacker performs the side-channel attack to leak the victim's memory access pattern.

This range-query attack improves the attack resolution and analysis correctness compared to the SQL execution attack (Figure 12): (1) in the range query attack, the attacker probes the memory in a per-set order, i.e., the attacker triggers the victim SQL query repeatedly and probes different NVCache sets for each repeat. Compared to the previous per-iter probing that sequentially probes all NVCache sets during each

```
-- Query records using a ranged condition
SELECT * FROM info WHERE create_date BETWEEN 06-01-2005 AND 07-01-2005;
```

(a) Ranged query (1st month)




(b) 1st month (c) 2nd month

(d) 3rd month (e) 4th month

Figure 13: Memory access pattern of SQLite ranged query.

iteration, this per-set probing runs at a much higher frequency (256× faster) and retrieves information with a much higher resolution. (2) the range-query attack uses a two-dimension Pearson correlation as distance in the k-NN algorithm to categorize results, i.e., the attacker calculates the correlation for each cache set and uses the average correlation coefficients as k-NN distances. This improves the categorization quality because range query attacks generate much more iterations, leading to high-noise results when using the one-dimension correlation.

Figure 13b to Figure 13e shows the side channel results of four victim queries using different date ranges. The color represents the memory access latency, which further indicates the victim's memory access in the corresponding NVCache set: Darker color represents a higher attacker latency and indicates the victim has more memory accesses in the cache set. These figures show that these queries have different memory access patterns in terms of access addresses and timings. We run each ranged query 100 times and use 70% of the results as training samples for the k-NN algorithm. In this evaluation, 120 queries are tested and 86 of them are correctly categorized, resulting in a 72% accuracy.

## 5.2 PMDK Key-Value Store Leakage

We mount the NVCache-based side channel attack to leak information from NVRAM-optimized applications.

**NVRAM-aware applications.** NVRAM-aware applications are designed to leverage NVRAM features, including byte-addressability, persistence, and DRAM-like performance. These applications (1) issue memory load and store instructions to access NVRAM instead of relying on filesystem read/write APIs; (2) issue cache line flush and memory fence instructions to flush data to NVRAM for persistence, instead of using conventional `fsync()`; and (3) configure the filesystem to bypass the DRAM page cache, thus ensuring CPU data is directly flushed to NVRAM, without relying on `fsync()` to flush the page cache.

It is non-trivial to implement such applications from scratch as the programmer has to manually insert the flush and fence instructions to proper code sites. Thus, programmers typically rely on NVRAM programming libraries that abstract these low-level operations and provide library functions to reduce the NVRAM programming complexity. PMDK [32] is one of the most mature and widely-adopted NVRAM programming libraries.

In this attack, we leak information from the applications that use PMDK to access NVRAM.

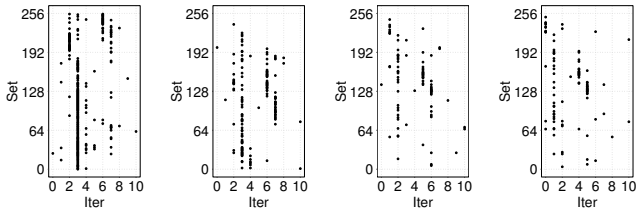
**Threat model.** The Threat model is similar to the database operation leakage (§ 5.1). The major difference is the victim in this attack is an NVRAM-aware application, which inherently flushes data to NVRAM at a high frequency to maintain fine-granularity crash consistency. This frequent memory access pattern enables attackers to detect more detailed information compared to the conventional filesystem access approach (§ 5.1).

**Leak PMDK KV store accesses.** We target an example key-value store application provided by PDMK developers [29]. This example uses the libmemobj-cpp library [28] and stores key-value pairs in NVRAM. To access them, it uses the PMDK library to achieve per KV-pair persistence and crash consistency. We mount the NVCache-based side-channel attack on this KV-store and leak which specific key-value pair is being accessed: Figure 14 shows four example memory access patterns that are visually distinguishable. These patterns can be categorized using the k-NN categorization described in § 5.1. We evaluated 8 different key-value updates, where each operation is executed 100 times and 70% of the results are used as the k-NN training set. In this evaluation, 182 out of 240 samples are correctly categorized, resulting in a 76% accuracy.

In conclusion, by targeting the NVRAM-aware victim applications that directly access the NVRAM, the attacker can detect a detailed NVRAM access pattern that is not filtered by the DRAM page cache, thus further learning high-resolution victim activities.

## 5.3 Code Execution Path Leakage

Motivated by the NVRAM memory mode (Figure 1), this section describes an attack that leaks code execution paths, where the attacker leverages the L1 NVCache to detect which shared library function is being executed by the victim.




(a) Key = 256 (b) Key = 512 (c) Key = 768 (d) Key = 1024

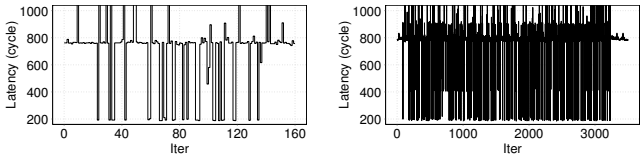
Figure 14: PMDK key-value store's memory access pattern when updating different key-value pairs.

```
for (int i=0; i<n; i++) {
    shared_lib_func();
    while(cnt < wait_cycle) {
        cnt++;
    }
}
```

(a) Victim code.

```
while (true) {
    cycle_start = rdtscp();
    tmp += *((char *)shared_lib_func);
    cycle_total = rdtscp() - cycle_start;
    clflush(shared_lib_func);
    flush_l1_nvcache();
}
```

(b) Attacker code.




(c) Temporal resolution (function call duration 22 us, interval 5 us)

(d) Detect wolfSSL private-key-dependent function calls, 225 out of 275 calls detected.

Figure 15: Code execution leakage and wolfSSL analysis.

**Threat model.** We assume the NVRAM is running in the memory mode (§ 2.1), acting as a large system memory instead of persistent storage. We assume the attacker and the victim share library code pages, similar to the Flush+Reload attack [84], and shared code pages reside in the NVRAM and are cached in the DRAM. The attacker repeatedly (1) flushes the shared library code pages to NVRAM, (2) waits for the victim's activities, (3) then reads the code pages back and times page loads. Whenever the victim calls a shared library function, the corresponding code page is loaded, which is further detected by the attacker as a low latency page load, thus leaking the victim's execution pattern of the shared library code.

**System configuration.** Instead of setting up the entire system to use NVRAM in the memory mode, we emulate this attack environment based on existing settings: The shared library is stored in an NVRAM-aware filesystem which enables direct access to the NVRAM and bypasses the DRAM page cache. This emulates the library code stored using the memory mode, just without the DRAM cache, as the attacker can follow prior approaches [20] to flush the code pages from DRAM. To achieve high temporal resolution, the attacker assumes the victim mostly runs on a certain CPU core, which is a realistic assumption. We emulate this setting by pinning the victim process to one CPU core.

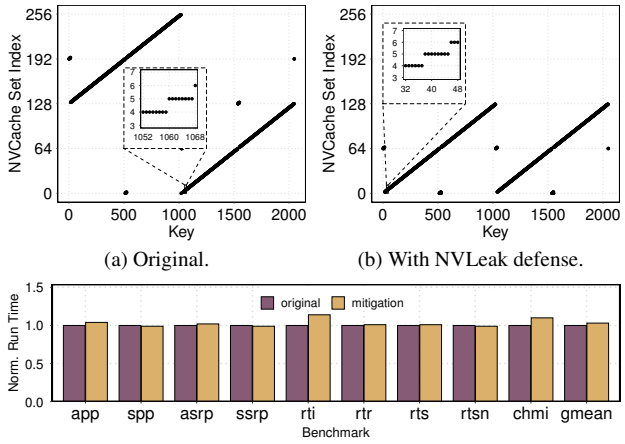
**Temporal resolution.** Figure 15a shows the victim code, which calls the library function, flushes the code page, and waits for predefined cycles. And Figure 15b shows the attacker code that times the library code page load, flushes the code page, and then flushes the L1 NVCache to prepare for the next iteration. Figure 15c shows that the attacker can detect victim function calls that take $\geq$ 22 us, with a call interval $\geq$ 5 us. The attacker can detect function calls shorter than 22 us with a risk of losing precision. This proof-of-concept shows that even if systems deploy mitigations for protecting on-chip side channels [43, 57], an attacker can still measure non-constant time code with relatively high spatial (256 B) and temporal resolution (22 us).

**Feasibility to attack cryptographic operations.** Based on the code execution leakage, we analyze the feasibility of attacking real cryptographic code, the wolfSSL library [25] version 4.2.0. We identified a secret-key-dependent branch in wolfSSL's `_fp_exptmod()` function as part of the RSA private key decoding process[3]. This function loops over the key material of an RSA private key using a sliding window algorithm which calls three extra functions in a set of secret-depend `if` statements, `fp_sqr()`, `fp_mul()`, and `fp_montgomery_reduce()`, depending on secret values. We further assume the victim is vulnerable to code page flushes [20]. Hence, the attacker can flush the shared code pages and apply the code execution analysis on these functions, to detect when secret-dependent branches are taken. The victim runs the full RSA decoding algorithm instead of just the isolated functions vulnerable to attacks.

Figure 15d shows the attacker's latency measurements of the `fp_sqr()` function, where low latencies indicate the victim calls of this function. In this example, the victim has called this function 275 times, and the attacker detected 225 calls. We repeat this attack 100 times with different secret keys and find that the attacker detects approximately 77% of the calls.

**Recovering RSA private key bits.** The detected function calls indicate that an attacker can recover significant parts of the key material from such non-constant-time implementations: The attacker monitors all three above-mentioned secret-dependent functions to learn: (1) the first bit of each 6-bit sliding window data, and (2) all the zero bits between sliding windows. This on average recovers 28.55% bits from 1000 randomly generated 2048-bit RSA keys. According to prior attacks exploiting partial information [7, 24, 46], a partial RSA key (with $\geq$ 27% bits recovered) has a high probability for an attacker to recover the full RSA key using the branch and prune technique [24].

[3] We disable wolfSSL's `harden` feature to attack the original version of this function, which is vulnerable to timing side channels.




(a) Original. (b) With NVLeak defense.

(c) Defense performance evaluation.

Figure 16: NVLeak mitigations applied to (a-b) PMDK key-value store and (c) a PMDK benchmark suite.

Table 2: PMDK Benchmarks in Figure 16c

| Name | Data Structure | Operation |
|---|---|---|
| app | Persistent Pointer | Assignment |
| spp | | Swap |
| asrp | Self-Relative Pointer | Assignment |
| ssrp | | Swap |
| rti | Radix Tree | Insert |
| rtr | | Remove |
| rts | | Search |
| rtsn | | Search (key not present) |
| chmi | Concurrent Hash Map | Insert |

## 6 Mitigations

We propose three mitigations based on reverse engineering results (§ 3) and attacks (§ 4 and § 5). We first propose a software-based partitioning mechanism to protect L2 NVCache and evaluate its effectiveness and performance. We then propose a software mitigation and a hardware mitigation to protect the NVCache hierarchy, based on prior works that are proved to be effective in defending against cache side-channels [13, 59, 61, 73, 75].

**Software-based mitigation for L2 NVCache.** The NVLeak attacks are made possible mainly because the performance of a security domain can be influenced by the memory accesses of another as they compete for shared on-DIMM resources such as L1 or L2 NVCaches. In particular, in our attacks, the victim and the attacker can compete for a single set in the L2 NVCache. Therefore, we propose a simple yet effective software-only mitigation scheme that isolates the L2 NVCache sets that belong to disjoint security domains. Since we know the exact indexing scheme used by the Optane DIMM (§ 3), we can modify the memory allocation such that the memory of a process can be mapped to specific sets in the L2 NVCache. This software mitigation leverages the set-associativity and thus only protects the L2 NVCache.

We implement the NVLeak mitigation at the persistent memory library level where all the persistent memory allocations are handled by PMDK [32]. We modify the PMDK's memory allocator, `make_persistent()`, to take an additional bitmap parameter that specifies a list of NVCache sets to be used. The allocator then allocates memory in the next block whose NVCache set is 1 in the bitmap. Hence, the allocator avoids the L2 NVCache sets that may leak information to an attacker.

We converted PMDK persistent data structures to use this secure memory allocator, and then converted the key-value store (§ 5.2) and a PMDK benchmark suite [28] to use the secure data structures.

Figure 16a shows how the memory allocated to different keys in the PMDK key-value store is mapped into different sets in the L2 NVCache. Without our defense, the memory allocation is unrestricted, and any key can be mapped into any NVCache set. However, when we enable our defense (Figure 16b), the memory allocator only allocates the memory regions that are guaranteed to be mapped into the lower sets in the L2 NVCache, eliminating any potential contention. We then perform the side channel attack described in § 5.2 and confirm that the attacker is not able to detect the victim's L2 NVCache activities when applying this mitigation. Figure 16c shows the performance evaluation of our defense using a PMDK benchmark suite. The full description of the benchmarks can be found in Table 2. On average, our partitioning mitigation incurs less than 4% performance overhead.

**Software-based mitigation for WPQ and L1 NVCache.** The WPQ and L1 NVCache leakage can be mitigated in software by flushing the WPQ and L1 NVCache before accessing secret data in NVRAM, or randomizing memory accesses in the program. This software mitigation can significantly reduce the possibility of leakages in these fully-associative structures [59], while the hardware mitigation is preferred to prevent these leakages at a lower performance cost [10, 75].

**Hardware-based mitigation.** To prevent information leakage in NVRAM, the hardware and software need the ability to distinguish memory requests from different security domains at the NVRAM DIMM level. This is similar to the Intel Cache Allocation Technology [27] (Intel CAT), which dynamically partitions CPU caches for different processes. Thus it's possible to extend the Intel CAT to partition the WPQ and NVCache, and provide a software interface for processes to allocate their private partitions. This prevents the sharing of these queue and cache structures and thus prevents the corresponding information leakages [13, 61, 73, 75].

## 7 Discussion

In this paper, we present the reverse engineering and attacks in NVRAM systems, demonstrate the results using Intel Optane DIMM, and propose mitigation designs. These techniques and ideas are generic and can be applied to investigate future hardware designs because: (1) Future NVRAM systems are likely to adopt similar design choices as Intel Optane DIMM, such as using on-NVRAM cache to fill the performance gap between fast interconnection and slow NVRAM media. These designs are likely to be vulnerable to off-chip attacks similar to the ones described in our paper. (2) Our tools do not rely on specific CPU instructions to access Intel Optane DIMM, thus can be used on future NVRAM devices as long as they support direct access through CPU memory load/store instructions (e.g., using CXL [12] technology). We hope our methodologies and tools influence the future NVRAM system design choices.

## 8 Related Work

To our knowledge, we are the first to develop covert and side-channel attacks on real NVRAM systems. This section discusses related works.

**NVRAM characterization.** Wang et al. [76] are the first to reverse engineer the Optane DIMM architecture with LENS, an NVRAM reverse engineering tool. That work revealed the NVRAM on-DIMM buffer sizes and their block sizes, but failed to reveal more detailed structures such as cache set sizes and indexing schemes. Zhang et al. [85] reverse engineered Optane DIMMs with GORDON – an FPGA-based profiling tool – and confirmed most of the LENS observations. Xiang et al. [79] investigated the on-DIMM buffer structures on the first and second generations of Optane DIMM and confirmed many observations from LENS [76]. Several recent studies [34,83] observed that repeated writes to a concentrated Optane DIMM memory area lead to high write latency. Compared to these prior works, our paper depicts more detailed NVRAM architectural designs that raise security concerns; our paper also revises several inaccurate conclusions from these previous works.

**NVRAM architecture security.** Liu et al. [45] presented a set of side-channel attacks on Optane DIMM, which is a concurrent work. Previous NVRAM architecture security works primarily focus on protecting NVRAM data integrity: Triad-NVM [5] proposed a new persistence mechanism of Merkle Tree and encryption counters to enable secure recovery of NVRAM data. SuperMem [87] facilitated a write-through counter cache to guarantee the security and atomicity of data writes at a low overhead. Freij et al. [17] proposed a set of optimizations to reduce the overhead of persisting the Bonsai Merkle Tree that is used to encrypt the NVRAM data. Xu et al. [81] proposed a framework to mitigate persistent memory object corruption vulnerabilities.

**Off-chip microarchitectural side channels.** Most previous attacks that bypass cache side-channel defense focus on on-chip components [14,33,51,63,72,74,82], but our work is an off-chip attack that directly affects the applications running on the CPU, so we will discuss such attacks further. Pessl et al. [55] showed that the DRAM row buffer could be used to spy on memory operations. However, a single row buffer with 8 KB spatial resolution is shared across many workloads on the system, which makes attacks relying on them highly noisy and impractical. Gruss et al. [20] demonstrated the page-cache attack, which essentially exploits the contention of memory pages within the DRAM. Page-cache attacks can be combined with NVLeak when NVRAM is used in memory mode. On the other hand, page-cache attacks are noisier than attacks based on the set-associative L2 NVCache and have a lower spatial resolution than the L1 NVCache with 256 B granularity. Aside from the off-chip DRAM, attackers can also exploit accelerators such as GPU [49] and FPGA [77] connected to the memory subsystem over PCIe. A major limitation of these attacks compared to NVLeak is that the memory shared by these accelerators and the CPU is generally not as privacy/security-sensitive as the file system and code pages, which are backed by NVRAM.

## 9 Conclusion

We conclude that NVLeak is a practical concern for the security of NVRAM storage. We demonstrate that attacks based on NVRAM's cache structure can break cross-core and cross-VM isolation and leak information from real-world applications such as databases, NVRAM-aware key-value stores, cryptography libraries, and more; We also demonstrate a leakage channel based on NVRAM-specific wear-leveling mechanism, which breaks the filesystem isolation and leaks Linux file metadata updates. Our empirical attack study motivates the development and deployment of new defense against NVRAM-based microarchitectural side channels, as we also came up with such mitigation. Ultimately, we hope that our work encourages the community to investigate more into the security of NVRAM and, in general, more holistic approaches to prevent side channels on complex computing systems.

## Acknowledgements

The authors thank the anonymous shepherd and reviewers for their careful feedback and support. This paper is supported in part by NSF grants 1829524, 2011212, and SRC/DARPA Center for Research on Intelligent Storage and Processing-in-memory.

## References


[1] Intel® Optane™ Persistent Memory. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *ATC*, 2008.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun and Profit. In *S&P*, 2019.

[4] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 1992.

[5] Amro Awad, Mao Ye, Yan Solihin, Laurent Njilla, and Kazi Abu Zubair. Triad-NVM: Persistency for integrity-protected and encrypted non-volatile memories. In *ISCA*, 2019.

[6] Naomi Benger, Joop van de Pol, Nigel P Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *CHES*, 2014.

[7] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *CHES*, 2017.

[8] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*, 2019.

[9] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant CPUs. In *CCS*, 2019.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 2019.

[11] Centers for Medicare & Medicaid Services. NPPES Dataset. URL: https://download.cms.gov/nppes/NPI_Files.html.

[12] CXL Consortium. Compute Express Link. URL: https://www.computeexpresslink.org/.

[13] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 2020.

[14] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems. In *ISCA*. IEEE, 2021.

[15] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[16] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[17] Alexander Freij, Shougang Yuan, Huiyang Zhou, and Yan Solihin. Persist level parallelism: Streamlining integrity tree updates for secure persistent memory. In *MICRO*, 2020.

[18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[19] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *CCS*, 2018.

[20] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. In *CCS*, 2019.

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[22] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, 2016.

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[24] Nadia Heninger and Hovav Shacham. Reconstructing rsa private keys from random key bits. In *CRYPTO*, 2009.

[25] WolfSSL Inc. WolfSSL embedded ssl/tls library. URL: https://www.wolfssl.com/.

[26] Intel. Intel 64 and IA-32 architectures software developer manual.

[27] Intel. Introduction to cache allocation technology in the Intel Xeon processor E5 v4 family.

[28] Intel. Libpmemobj-cpp: C++ bindings and containers for libpmemobj. URL: https://github.com/pmem/libpmemobj-cpp.

[29] Intel. Libpmemobj-cpp examples. URL: https://github.com/pmem/libpmemobj-cpp/tree/master/examples/map_cli.

[30] Intel. 2nd generation Intel® Xeon® Scalable processors with Intel® C620 series chipsets (purley refresh), 2019.

[31] Intel. Intel Optane DC Persistent Memory, 2019. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.

[32] Intel. Persistent memory development kit, 2019. URL: https://pmem.io/.

[33] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *ASIA CCS*, 2016.

[34] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv*, 2019.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[36] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *S&P*, 2020.

[37] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA*, 2009.

[38] Linux Kernel Organization. Direct access for files. URL: https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[39] Linux Kernel Organization. ext4 Data Structures and Algorithms. URL: https://www.kernel.org/doc/html/latest/filesystems/ext4.

[40] Linux Kernel Organization. KVM-unit-tests. URL: https://www.linux-kvm.org/page/KVM-unit-tests.

[41] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical keystroke timing attacks in sandboxed javascript. In *European Symposium on Research in Computer Security*, 2017.

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[43] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 2016.

[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *S&P*, 2015.

[45] Sihang Liu, Suraaj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, , and Samira Khan. Side-channel attacks on optane persistent memory. In *USENIX Security*, 2023.

[46] Gabrielle De Micheli and Nadia Heninger. Recovering cryptographic keys from partial information, by example, 2020.

[47] Micron Technology, Inc. 3D XPoint technology, 2018. URL: https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[48] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 2019.

[49] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: Gpu side channel attacks are practical. In *CCS*, 2018.

[50] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA*, 2006.

[51] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. Lord of the ring (s): Side channel attacks on the CPU on-chip ring interconnect are practical. In *USENIX Security*, 2021.

[52] Colin Percival. Cache missing for fun and profit. In *In Proc. of BSDCan 2005*, 2005.

[53] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. Make sure DSA signing exponentiations really are constant-time. In *CCS*, 2016.

[54] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for Cross-CPU attacks. In *USENIX Security*, 2016.

[55] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security*, 2016.

[56] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *S&P*, 2021.

[57] Moinuddin K Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, 2018.

[58] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *USENIX Security*, 2021.

[59] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security*, 2015.

[60] Andy M Rudoff. Deprecating the PCOMMIT Instruction, 2016. URL: https://software.intel.com/en-us/blogs/2016/09/12/deprecate-pcommit-instruction.

[61] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design. In *USENIX Security*, 2021.

[62] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *CCS*, 2019.

[63] Johanna Sepúlveda, Mathieu Gross, Andreas Zankl, and Georg Sigl. Beyond cache attacks: Exploiting the bus-based communication structure for powerful on-chip microarchitectural attacks. *ACM Transactions on Embedded Computing Systems (TECS)*, 2021.

[64] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. Database reconstruction from noisy volumes: A cache Side-Channel attack on SQLite. In *USENIX Security*, 2021.

[65] SQLite Consortium. SQLite. URL: https://www.sqlite.org/index.html.

[66] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. SecSMT: Securing smt processors against contention-based covert channels. In *USENIX Security*, 2022.

[67] Daniel Townley and Dmitry Ponomarev. SMT-COP: Defeating side-channel attacks on execution units in smt processors. In *PACT*, 2019.

[68] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *USENIX Security*, 2018.

[69] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 2020.

[70] Jose Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Chris Fletcher, and David Kohlbrenner. Augury: Using data memory-dependent prefetchers to leak data at rest. In *S&P*, 2022.

[71] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011.

[72] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on cpu mesh. In *S&P*, 2022.

[73] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In *DAC*, 2016.

[74] Yao Wang and G Edward Suh. Efficient timing channel protection for on-chip networks. In *International Symposium on Networks-on-Chip*, 2012.

[75] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.

[76] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *MICRO*, 2020.

[77] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. Jackhammer: Efficient rowhammer on heterogeneous fpga-cpu platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

[78] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, 2019.

[79] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-DIMM buffering. In *EuroSys*, 2022.

[80] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, 2016.

[81] Yuanchao Xu, Chencheng Ye, Xipeng Shen, and Yan Solihin. Temporal exposure reduction protection for persistent memory. In *HPCA*, 2022.

[82] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 2019.

[83] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *FAST*, 2020.

[84] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[85] Jialiang Zhang, Nicholas Beckwith, and Jing Jane Li. GORDON: Benchmarking Optane DC persistent memory modules on FPGAs. In *FCCM*, 2021.

[86] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*, 2020.

[87] Pengfei Zuo, Yu Hua, and Yuan Xie. SuperMem: Enabling Application-transparent Secure Persistent Memory with Low Overheads. In *MICRO*, 2019.

# A Additional Reverse Engineering Results

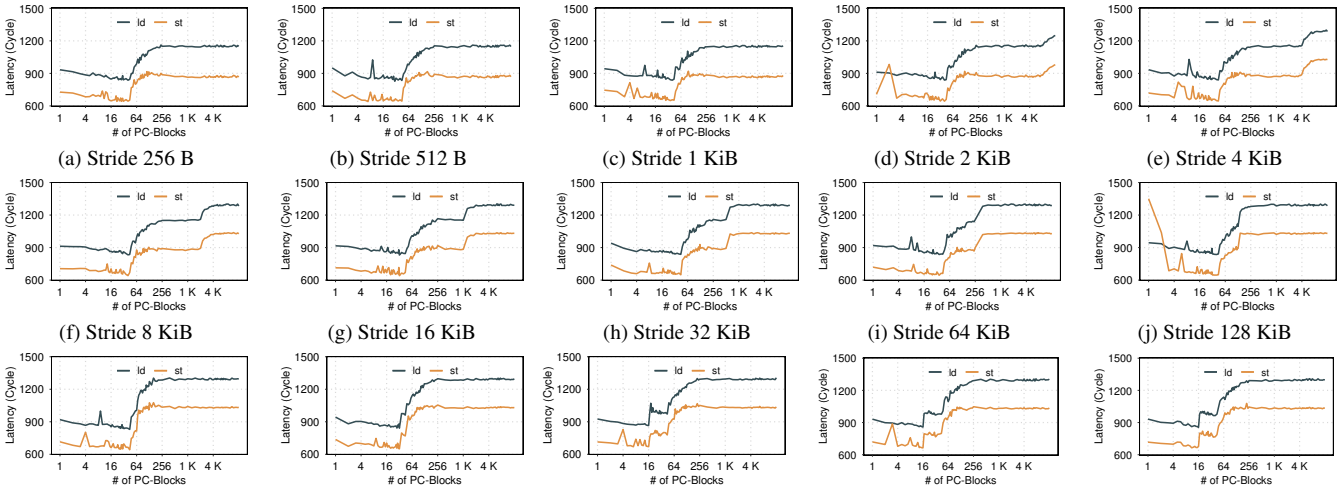
## A.1 Additional Results on Server A

We present the majority of the Server A (Table 1) reverse engineering results in the main paper, this section provides additional results of NVCache reverse engineering: Figure 17a to Figure 17o are NVLeak's strided pointer chasing results, which are detailed illustrations of the Figure 4.

## A.2 Additional Results on Server B

In this section, we provide reverse engineering on an additional server machine (Server B from Table 1) which equips Optane DIMMs of different sizes and firmware versions compared to Server A.

Figure 18 shows the NVLeak reverse engineering results. Following the analysis in § 3, we conclude that Server B's Optane DIMMs have similar architecture design and performance characteristics as in Server A: Server B Optane DIMM has two levels inclusive NVCaches (Figure 18a), where L1 is 16 KiB fully associative cache with 256 B blocks, and L2 is a 16-way set associative cache with 4 KiB blocks. This Optane DIMM also has a long latency effect triggered by repeated writes (Figure 18c), and this latency is timing-sensitive (Figure 18b and Figure 18d). This Optane DIMM's data-migration trigger conditions (Figure 18e) under 256 B region have a median of 2,802 write iterations, with 50% of them falling in the `[2782:2814]` range, which is much more concentrated than in Server A (Figure 7).

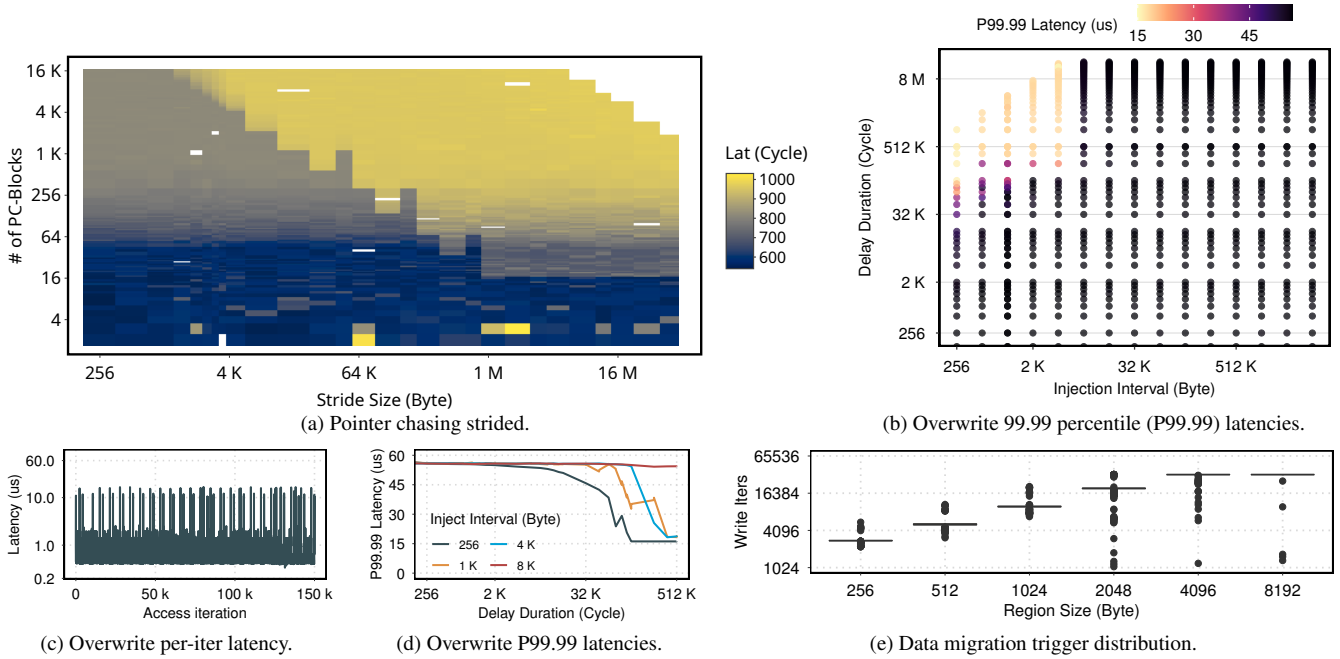



(a) Stride 256 B (b) Stride 512 B (c) Stride 1 KiB (d) Stride 2 KiB (e) Stride 4 KiB

(f) Stride 8 KiB (g) Stride 16 KiB (h) Stride 32 KiB (i) Stride 64 KiB (j) Stride 128 KiB

(k) Stride 256 KiB (l) Stride 512 KiB (m) Stride 1 MiB (n) Stride 2 MiB (o) Stride 4 MiB

Figure 17: Pointer chasing latency under various stride sizes.




(a) Pointer chasing strided. (b) Overwrite 99.99 percentile (P99.99) latencies.

(c) Overwrite per-iter latency. (d) Overwrite P99.99 latencies. (e) Data migration trigger distribution.

Figure 18: Reverse engineering results on Server B (Table 1).