Peek-a-Walk: Leaking Secrets via Page Walk Side Channels

Alan Wang
UIUC
alanlw2@illinois.edu

Boru Chen
UC Berkeley
boruchen@berkeley.edu

Yingchen Wang
UC Berkeley
yingchenwang@berkeley.edu

Christopher W. Fletcher
UC Berkeley
cwfletcher@berkeley.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

David Kohlbrenner
University of Washington
dkohlbre@cs.washington.edu

Riccardo Paccagnella
Carnegie Mellon University
rpaccagn@cs.cmu.edu

Abstract—**Microarchitectural side-channel attacks are an insidious threat to program security. An emerging class of these attacks constructs gadgets that dereference the contents of data memory directly. This is caused by optimizations, such as speculative execution and data-memory prefetching, that can guess (incorrectly) that the program is performing a pointer chase. In theory, this is devastating for security, as dereferencing a secret seemingly leaks it over memory-based side channels, e.g., through the cache. In practice, it is not. Since most secrets do not look like valid pointers, their dereference typically fails and does not leak anything.**

In this paper, we introduce the page walk side channel (PWSC), a new attack that can leak information even when an invalid pointer is dereferenced. In particular, given a 64-bit secret that passes the address canonicality check, PWSC can leak all remaining bits of the secret except for the low-order 6 bits, without making any assumptions on what these bits look like. We demonstrate how PWSC amplifies leakage in scenarios exploiting speculative execution and data-memory prefetching. For speculative execution, we show that PWSC, combined with Intel's LAM feature, can be exploited to leak nearly all of physical memory and that even without LAM, PWSC can be used to leak Dilithium secret keys. For data-memory prefetching, we reverse engineer the semantics of Intel's data-memory dependent prefetcher (DMP) and show how this DMP and PWSC can be combined to break security in an intra-process sandbox setting.

1. Introduction

The past few years have marked the beginning of a new era in microarchitectural side-channel attack research. Traditionally, adversaries were limited to leaking information from architecturally executed operations [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. In this new era, however, adversaries can leak information from operations that the victim had never intended to execute but were still executed microarchitecturally as a result of CPU optimizations. A notable example of such an operation occurs when speculative execution or data prefetchers perform a direct dereference of secret data (i.e., `*secret`). Recent work demonstrates that this dereferencing operation can be exploited to increase the attack surface of Spectre [14] and break constant-time cryptographic implementations [15].

Fortunately, there has been a saving grace, namely that it is difficult for an adversary to leak sensitive information from `*secret` when `secret` contains arbitrary 64-bit data [16], [17]. Suppose `*secret` passes the address canonicality check—meaning that its high-order 16 bits (i.e., bits 63:48) are copies of its 47th bit—or that mechanisms are enabled that relax this canonicality check [14]. Ideally, an adversary would like to be able to leak the remaining 47 bits, regardless of the distribution they follow. However, this is not possible today. Case in point, attacks exploiting speculative executions of `*secret` can only leak the low-order 47 bits when `secret` is ASCII data with a known prefix, limiting targets to only the root password hash [14]. Similarly, attacks exploiting prefetcher-induced executions of `*secret` leak information only when `secret` is manipulated to pass specific "looks-like-a-pointer" heuristic checks, which requires bespoke cryptanalysis [15].

This paper ameliorates the above issues by introducing the page walk side channel (PWSC). PWSC demonstrates that an adversary observing the execution of `*secret` can leak up to 42 bits of `secret` (specifically, bits 47:6)—without making any assumptions on what these bits look like. To our knowledge, PWSC exceeds the bit leakage capability of and operates under fewer assumptions than all previously known memory-based side channels. For example, existing cache side channels can only leak the bits of `secret` overlapping with the cache set index bits (i.e., bits 11:6) [1], [2], [5], [8], [18], [19], [20], [21] or, at best, some of the cache line offset bits (i.e., bits 5:2) [10]—if and only if `secret` represents a valid (mapped) virtual address.

PWSC exploits the cache side-channel leakage inherent to the page walk process. When `*secret` executes and induces a page walk, the MMU uses bits 47:12 of `secret` as indexes into four levels of page tables, and this process results in cache fills that encode parts of these bits [17]. Decoding these bits is, however, nontrivial because programs also perform unwanted page walks, and each page walk causes multiple, potentially-aliasing cache fills. This makes it difficult to observe the secret-dependent page walk and reconstruct which cache fill is due to which bits of `secret`. PWSC relies on two novel components that enable an adversary to overcome these challenges:

1) PWSC's first component is a *differential Prime+Probe* receiver (Section 3.1) which can capture and isolate the cache side effects of a page walk on `secret` even in the presence of systematic noise, i.e., reproducible noise that occurs across experiments. This technique leverages the fact that `*secret` only executes microarchitecturally as a result of a CPU optimization. Then, the key idea in differential Prime+Probe is for the adversary to run two identical experiments. In one experiment, the adversary allows the optimization to dereference `*secret`. In the other they do not, e.g., by not completing a predictor's training procedure. The adversary can then isolate the page walk due to `secret` by subtracting the two signals.
2) PWSC's second component is an *order oracle* (Section 3.2) that can map each page table access inferred from the cache state to its corresponding page table (PT) level, arranging the bits recovered from the cache state into their correct order. In particular, PWSC's memory mapping order oracle selectively maps memory to progressively increase the page table level where the page walk on `secret` terminates. This way, only one new page table index is revealed at a time, which is sufficient for the adversary to disambiguate which cache fill corresponds to which PT index.

We demonstrate the security implications of PWSC through two case studies that explore how PWSC can increase information leakage over the state of the art through distinct microarchitectural optimizations that are capable of dereferencing invalid pointers (potentially-secret data).

The first case study explores how to enhance Spectre-V2 attacks with our PWSC receiver, with and without future hardware features. Under this threat model, we show that an adversary can leak almost all of physical memory with future hardware features enabled. Moreover, PWSC can be used to leak secret keys from the post-quantum Dilithium cryptosystem even without said future hardware features.

The second case study explores how to enhance leakage through Intel's data-memory dependent prefetcher (DMP) using PWSC. While prior work identified the existence of this DMP and determined an activation pattern for it [15], we are the first to reverse engineer its semantics and show that any DMP can be used to leak invalid virtual addresses. We then show how this DMP and PWSC can be combined to break security in an intra-process sandbox setting.

In summary, this paper contributes the following:

1) We introduce the page walk side channel (PWSC), a new type of side channel and associated attack techniques that, by observing the execution of `*secret`, can reconstruct up to 42 bits of `secret`—without making any assumptions on what these bits are and even when `secret` does not represent a valid address.

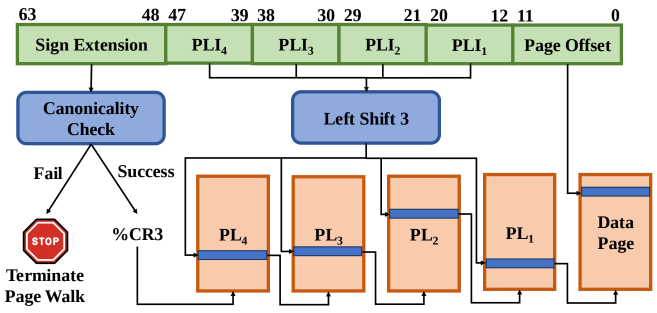

Figure 1. A page walk on a virtual address begins if and only if the address passes the canonicality check. During the page walk, each PT index is used to select the PTE with the address of the next level's PT. This repeats until the translation fails or when the physical address of the data page is found. The left shift 3 is performed to adjust for the size of each PTE (8 bytes).

2) We combine PWSC with existing Spectre-V2 attacks to demonstrate that prior work underestimated the danger of unmasked Spectre gadgets in the kernel. We demonstrate that, with Intel LAM enabled, PSWC can leak almost all of physical memory and, without Intel LAM, PWSC can still leak valuable cryptographic keys.
3) We reverse engineer the semantics of the Intel DMP and show that PWSC, combined with new Intel DMP-specific attack techniques, can extend the leakage capabilities of DMP attacks and break isolation in an intra-process sandbox setting.

Disclosure. We disclosed our findings to Intel and met with Intel researchers to discuss the issue in Q2 2024. Intel referred to existing mitigations, such as Branch History Injection mitigations and the upcoming Linear Address Space Separation (LASS) feature (cf. Section 6). Intel also updated their public guidance on these mitigations [22].

2. Background

2.1. Page tables & page walks

Modern processors employ 64-bit virtual addresses and either 48-bit or 57-bit virtual address spaces, resulting in 4 or 5 levels of page tables (PTs) respectively. W.l.o.g. we will assume 48-bit virtual address spaces and 4 levels of PTs for the rest of the paper. The remaining upper bits of a 64-bit virtual address must be the sign extension of the 47th bit[1] to pass a *canonicality* check before the page walk process begins. If the check fails, no page walk occurs. If the action triggering the page walk was an instruction and that instruction becomes non-speculative, an exception occurs. If the 47th bit is 0, the virtual address is in user space and called a userspace address / pointer; otherwise, it is in kernel space and called a kernel-space address / pointer.

The 48 bits of a virtual address are split between the virtual page number (VPN) and the page offset. We will

1. That is, the most significant bit of the virtual address that is a part of the virtual address space. We index starting from 0 throughout the paper.

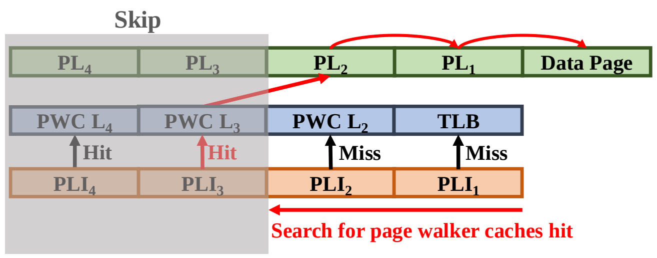

Figure 2. Page walker caches (PWCs) contain partial translation of VPN bits. In the event of a hit in the page walker caches, the MMU skips as many PT accesses as possible to improve performance. For example, when there is a hit in the 3rd level page walker cache, the MMU skips accessing PL_4 and PL_3 and directly moves to the PL_2 access.

assume 4 KB data and PT pages. Hence, the VPN is 36 bits and the page offset is 12 bits. Each PT page is made up of 8-byte page table entries (PTEs), resulting in 512 PTEs per PT page. The VPN bits are split into 4 *PT indexes* (9 bits per PT index) that select which PTE in each PT level.

We refer to the PT at each level as PL_4, PL_3, PL_2, PL_1 and a VPN's PT indexes into each level as PLI_4, PLI_3, PLI_2, PLI_1, respectively. During the page walk, the memory management unit (MMU) walks the PT tree starting from PL_4. The register CR3 points to PL_4 and uses PLI_4 to index into PL_4 to obtain PL_3, and so on. This process terminates when the MMU reaches the last PT where PLI_1 returns the PTE that gives the physical page number, which is used to access the data page. The page offset bits require no translation as they are shared between the virtual and physical addresses. This process can be seen in Figure 1.

Early terminating page walk. If a PT is not present in memory, the MMU will trigger a page fault. This can occur at any level of the PT tree and will terminate the page walk early. Specifically, for $i > 1$ the MMU may look up PL_i at PLI_i but terminate without looking up PL_{i-1}.

TLB & page walker caches. To improve page walk performance, each CPU core of most modern processors employs four levels of *page walker caches* (also known as *translation caches* or *paging-structure caches*) [23], [24], [25], [26], [27]. Each level of the page walker caches is referenced using the bits of the VPN up until the end of that level's PT index and stores the physical address of the next PT level. Specifically, given a virtual address, the 4th level page walker cache is referenced using bits 47:39, the 3rd level using bits 47:30, the 2nd level using bits 47:21, and the 1st level (also known as the *TLB*) using bits 47:12 (i.e., the entire VPN). Before starting a page walk on a virtual address, the MMU checks if the various page walker caches (starting from the TLB) contain a full or partial translation of that address' VPN bits. If there is a cache hit, the MMU uses the cached translation to skip as many PT accesses as possible, as can be seen in Figure 2. For example, if there is a hit in the 3rd level page walker cache, the MMU skips accessing PL_4 and PL_3 and directly moves to the PL_2 access.

Supervisor mode access prevention (SMAP). SMAP is a processor feature that, when enabled, prevents the processor from accessing userspace memory from kernel mode. As pointed out by prior work [14], SMAP blocks the final access to user data (and its associated cache fills) but only after completing that access' page walk and associated fills to the TLB and the page walker caches.

2.2. Caches and Cache Side Channels

Cache architecture. Modern processors use a cache hierarchy to reduce memory access latency. Typically, higher-level caches are smaller and faster to access, while lower-level caches are larger but slower to access. For example, the Intel processors we study in this paper have three cache levels, a core-private L1/L2 and a shared L3. These caches are set-associative, meaning that they contain a fixed number of cache sets, each of which can fit a fixed number of cache lines. Cache lines are the basic unit for cache transactions. Cache line size on Intel processors is typically 64 bytes, resulting in eight 8-byte PTEs per cache line.

Cache side-channel attacks. In a cache side-channel attack, an attacker infers a victim program's secret by observing the side effects of the victim program's secret-dependent accesses to the processor cache. These attacks typically consist of three steps, during which the attacker (i) brings the cache into a known state, (ii) lets the victim execute, and (iii) checks the state of the cache to learn information about the victim's execution during step (ii).

There are two popular styles of cache side-channel attacks: Flush+Reload [3] and Prime+Probe [1], [2]. In Flush+Reload, the attacker shares the target cache line with the victim. The attacker first flushes the target cache line to memory and then measures its access latency to deduce if the victim has accessed it. In Prime+Probe, the attacker builds an eviction set of addresses that map to a target cache set, primes the cache set with the eviction set, and probes it to figure out whether the victim accessed that target cache set and displaced a line in the eviction set. Prime+Probe does not require the attacker and victim to share memory. Our attacks leverage on Prime+Probe on the L1 cache.

2.3. Information leakage through page walks

Gras et al. [17] show how the page walk process leaks information over the cache side channel. Their observation is that during the page walk, the MMU brings accessed PTEs into the L1 data cache. The respective cache fills associated with each PTE leak part of their respective PT index. Using parameters from before, there are 8 PTEs per 64-byte L1 cache line. Given a 9-bit PT index used to select a PTE, the low-order 3 bits are a part of the cache line offset and the high-order 6 bits are used to select the cache set. An attacker monitoring the L1 cache state can therefore learn the high-order 6 bits of the PT indexes, as shown in Figure 3. Gras et al. exploited this leakage to break address space layout randomization (ASLR), assuming that the attacker

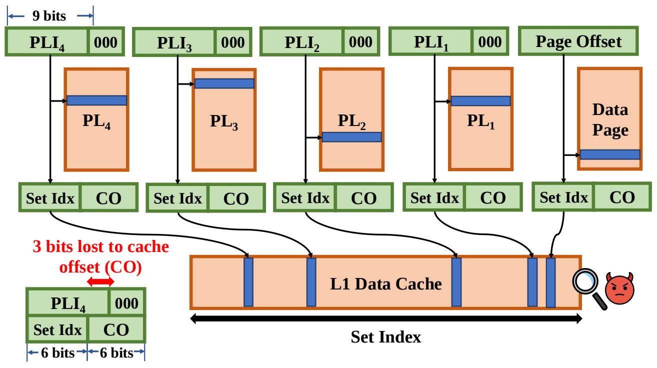

Figure 3. During a page walk, the MMU brings touched PTEs into the L1 data cache. The most significant 6 bits of a PT index determine the cache set accessed during its respective cache fill. The remaining 3 bits are part of the cache line offset.

has control over the valid address that undergoes translation during the page walk. The next section demonstrates an attack in a weaker setting—where the attacker does not control the address being translated—and shows how it can leak information about data that is an invalid address.

2.4. Spectre attacks

Spectre attacks exploit how a processor can microarchitecturally execute instructions outside of the program's architectural semantics. These *speculative* instructions can access and transmit secrets over microarchitectural side channels (e.g., the cache), which can then be measured by the attacker (the 'receiver'). Researchers have leveraged a number of microarchitectural structures to induce speculation (e.g., the branch target buffer) which gives rise to a number of Spectre variants [28], [29], [30], [31], [32], [33], [34], [35], [36], [37]. In this paper, we focus on the branch history injection (Spectre-BHI) variant [29], [38]. We provide background on Spectre-BHI in Section 4.2.

2.5. Data memory-dependent prefetchers

Data memory-dependent prefetchers (DMPs) are a class of hardware prefetchers that are optimized to reduce average memory access time when the program has an indirect (e.g., pointer-chasing) access pattern. Prefetchers are conceptually next-in-sequence predictors. Classical prefetchers try to predict the next-in-sequence by looking at the memory address trace. DMPs, on the other hand, try to predict the next-in-sequence by looking at the memory address trace and the contents of memory. This increases the information leakage possible through a DMP. There have been several proposed types of DMPs [15], [39], [40], [41], [42], [43], [44], [45], [46] that differ in what specific pattern they prefetch. However, while prior work reverse engineered the activation patterns of the DMP deployed in modern Apple processors [15], [42], little is known about the DMP deployed in Intel processors besides its existence and a basic activation pattern for it [47].

3. Page walk side channel (PWSC)

In this section, we introduce the *page walk side channel* (PWSC). We demonstrate that, given a memory dereference of an arbitrary 64-bit data value, PWSC can leak up to 42 bits of that 64-bit data value, exceeding the bit leakage capability of previously known memory-based side channels.

Notation and assumptions. Like prior microarchitectural side channels, PWSC involves a transmitter in the victim's security domain and a receiver in the attacker's security domain. PWSC's transmitter performs a memory dereference of a secret 64-bit data value. We refer to this memory dereference as `*secret`. We do not make any assumptions on the value being dereferenced. In particular, `secret` does not need to be a valid virtual address. We assume that the attacker can influence whether `*secret` occurs in the victim. We also assume that the attacker can cause `*secret` to induce a page walk, and that the attacker can monitor the side effects of this page walk on the CPU's L1 cache. Going forward, we refer to a page walk on `secret` as a *secret-dependent page walk*.[2] Sections 4 and 5 discuss threat models where these assumptions hold in practice.

Overview. Recall from Section 2.1 that when `*secret` executes and induces a page walk, `secret`'s VPN bits (i.e., bits 47:12) are used by the MMU as indexes into four levels of PTs, and, if the page walk succeeds, `secret`'s 12 least significant bits (i.e., bits 11:0) are used as the offset into a data page to complete the dereference. The goal of PWSC's receiver is to recover as many of the bits of `secret` that are used during this process (i.e., bits 47:0) as possible. To this end, PWSC's receiver uses two novel mechanisms:

1) A *cache receiver* that can observe the effects of a secret-dependent page walk on the cache (Figure 3) even in the presence of systematic noise.
2) An *order oracle* that can map each access to its corresponding PT level, arranging the bits recovered by the cache receiver into their correct order.

Next, we describe these mechanisms and demonstrate that PWSC can leak up to 42 bits of `secret`, corresponding to its entire VPN bits (47:12) and half of its page offset bits (11:6)—without making any assumptions on what is in these bits and even when `secret` is not a valid virtual address. To our knowledge, PWSC exceeds the bit leakage capability of all previously known memory-based side channels.

Experimental setup. We run all this paper's experiments on the P-cores of an Intel Core i9-13900K processor, featuring a 12-way and 64-set L1 cache. Our machine has 16 GB of RAM and runs the Ubuntu 22.04 OS with kernel 6.6-rc4.

2. By secret-dependent page walk we mean a page walk that happens unconditionally on the value of `secret`, resulting in PT accesses that encode the bits of `secret`. This is not to be confused with a page walk that occurs conditionally on the value of `secret`, which would only leak whether `secret` matches the branch's condition.

3.1. PWSC cache receiver

PWSC's cache receiver uses what we call the *differential Prime+Probe* technique. Recall from Section 2.2 that Prime+Probe relies on priming the cache, letting the victim (in our case, `*secret`) execute, and probing the cache to capture the side effects of the victim's execution. For `*secret` to result in a secret-dependent page walk, PWSC also requires evicting `secret` from the TLB. Further, to capture the full page walk signal via the cache side channel, PWSC needs to flush the page walker caches, which could potentially cause the page walk to skip accessing certain PT levels (cf. Section 2.1). Unfortunately, these extra eviction steps introduce significant noise and result in false positives in the Probe phase. In the following, we describe the baseline PWSC cache receiver and the denoising techniques we employ to remove these false positives.

Baseline cache receiver. PWSC's baseline cache receiver consists of two high-level steps: (i) first, we flush the TLB and page walker caches, and (ii) second, we perform Prime+Probe on the CPU's L1 cache. For step (i), our receiver employs cache thrashing. That is, we perform a large (empirically determined) number of page-walk inducing memory accesses that result in filling up the entire TLB and page walker caches. For step (ii), our receiver relies on a conventional L1 Prime+Probe [1], [2], generating eviction sets that cover the CPU's entire L1 cache.[3]

Combining the two steps, for each set of the L1 cache, the baseline receiver first flushes the TLB and the page walker caches; it then performs Prime+Probe on that set, monitoring the latency of each address in the target eviction set individually and recording the number of cache misses. Finally, it repeats this process for the next cache set.

We evaluate PWSC's baseline cache receiver on a synthetic `*secret` transmitter. For now, we embed this transmitter in the same program as the receiver and construct `secret` to represent a valid address (i.e., one for which each PT is present and the page walk does not fail). We run PWSC's receiver 128 times for each L1 cache set, executing `*secret` between the Prime and the Probe steps.

Figure 4 (Left) shows the results, indicating a high cache miss rate for many L1 cache sets. Recall from Figure 3 that a clean page walk involves five cache fills, corresponding to the four page table accesses and the final data page access. Hence, an ideal PWSC receiver should observe misses in no more than five L1 cache sets.[4] Unfortunately, our results indicate that the baseline PWSC is far from ideal and suffers from systematic false positives. That is, using the baseline PWSC receiver, the adversary is unable to determine the cache sets associated with the secret-dependent page walk.

3. Since the L1 is virtually indexed, constructing these L1 eviction sets is trivial. The adversary simply picks arbitrarily virtual addresses with the desired L1 cache set index bits (i.e., bits 11:6).

4. In the case of overlapping L1 cache sets the adversary would observe multiple misses in the same cache sets.

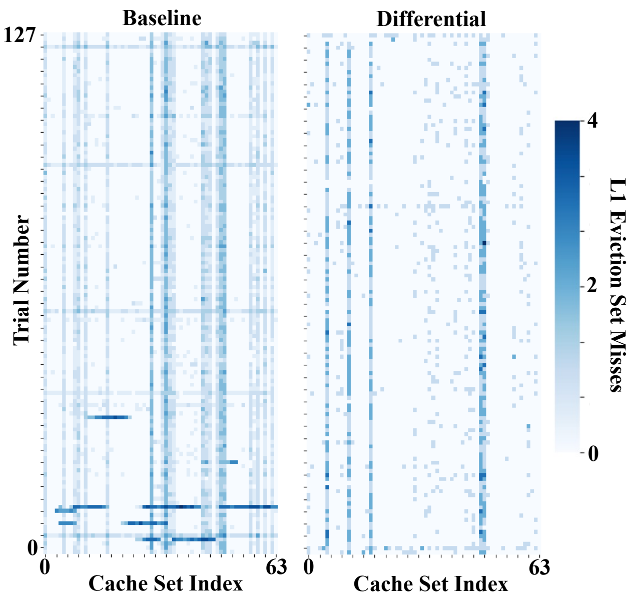

Figure 4. Left: baseline Prime+Probe cache receiver results. Right: differential Prime+Probe cache receiver results. Darker colors are associated with more L1 eviction set misses during the Probe step.

Differential Prime+Probe receiver. We now describe PWSC's differential Prime+Probe receiver. Recall that the baseline receiver starts by flushing the TLB and the page walker caches. This step is necessary for `*secret` to trigger a secret-dependent page walk and for this page walk to affect the cache. However, a side effect of this step is that it also triggers a page walk on every load performed during the attack. These page walks result in additional cache fills, which, as Figure 4 (Left) shows, introduce systematic false positives. To isolate the cache fills due to the secret-dependent page walk, PWSC's receiver needs a method to mask out the noise due to non-secret dependent page walks.

PWSC performs this denoising by augmenting the baseline cache receiver with *differential Prime+Probe*. This technique relies on running the baseline receiver twice. In the first run, the attacker influences the victim so that `*secret` occurs, observing the side-effects of both the secret-dependent page walk and the non-secret dependent page walks (as in the baseline receiver). In the second run, the attacker influences the victim so that `*secret` does not occur,[5] only observing the side-effects of the non-secret dependent page walks. The attacker can thus subtract the cache side-channel signal observed in the second run from the one observed in the first run to uniquely identify the cache sets associated with the secret-dependent page walk. Figure 4 (Right) shows the results of running PWSC's differential cache receiver 128 times for each L1 cache set. The results demonstrate that, using the differential PWSC receiver, the attacker is able to clearly determine the cache

5. As we show in Sections 4 and 5, this capability holds when `*secret` is performed only microarchitecturally and the attacker can influence the microarchitectural optimization to perform `*secret` selectively.

sets associated with the secret-dependent page walk (four page table accesses and the final data page access).

3.2. PWSC order oracle

PWSC's cache receiver can learn the L1 cache set indexes associated with a secret-dependent page walk. These indexes correspond to the high-order 6 bits of the four PT indexes in `secret`'s VPN bits and the high-order 6 bits of `secret`'s page offset bits (cf. Figure 3). To reconstruct the position of these bits within `secret`, however, the attacker also needs a way to map each cache set index to its corresponding PT level or data page access. That is, the attacker needs an *order oracle* that can order the five cache fills observed by PWSC's receiver. We now describe two methods to implement such an oracle. Additionally, we show that the first method can leak the low-order 3 bits of each PT index, enabling PWSC to leak up to 42 bits of `secret`.

Memory mapping order oracle. The first order oracle method involves selectively mapping memory in the adversary's virtual address space to cause the secret-dependent page walk to fail at specific, attacker-selected levels. This method is applicable in threat models where `secret` is allowed to map to the adversary's virtual address space. As we show in Sections 4 and 5, these include the user-to-kernel and the intra-process sandboxing threat models.

Using the memory mapping order oracle, the adversary moves what level of the secret-dependent page walk fails from higher levels to lower levels, one by one. This way, only one new PT index is revealed via the cache side channel at each round. By proceeding one level at a time, this order oracle allows to directly map each cache set index to its corresponding PT level or data page access.

Suppose bits 47:0 of `secret` are random and, as a result, `secret` does not represent a valid address. When `*secret` executes, the MMU attempts to perform a page walk on `secret` (Figure 1). The MMU accesses `CR3` to find PL_4's address. Next, it uses PLI_4 (bits 47:39 of `secret`) to select a PTE from PL_4, which requires a cache fill. However, since `secret` is not a valid address, the retrieved PTE is likely[6] going to be marked as not present. As a result, the secret-dependent page walk terminates. Further, since the page walk only involved the cache fill corresponding to PL_4's access, PWSC's receiver can directly leak the high-order 6 bits of PLI_4 (bits 47:41 of `secret`).

To continue the attack, the adversary needs a mechanism to mark the PTE retrieved from PL_4 as present and cause the secret-dependent page walk to fail at PL_3. We perform this step by creating a new mapping in the adversary's own address space at a virtual address that shares PLI_4 with `secret`. As a result of this new mapping, the above PTE now also covers the new adversary's address and is marked as present by the MMU. The secret-dependent page walk can thus move to PL_3, revealing the cache fill associated

6. See the following limitation when the PTE is a valid entry.

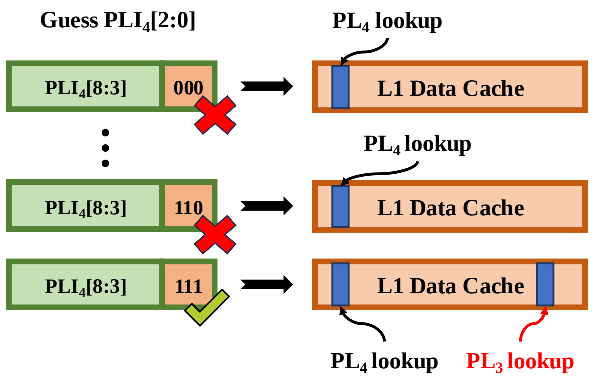

Figure 5. For each PT index the attacker guesses the low-order 3 bits and maps memory based on the guess. If the guess is correct, it will result in an extra cache fill due to next level's PT look up. In this figure, the attacker is brute forcing the least significant 3 bits of PL_4 as an example. When the guess is wrong (`000-110`), there is only an access to PL_4. However, when the guess is correct (`111`), there is an extra access to PL_3.

with PL_3's access, and only terminate again due to the not-present PTE retrieved from PL_3. The adversary can then use the same method to mark this new PTE as present and learn, one by one, the cache fills associated with all PT levels.

The above mechanism relies on the adversary creating a mapping at a virtual address that shares PLI_4 with `secret`. However, at this stage, the adversary only knows the high-order 6 bits of PLI_4. We address this issue by brute forcing the remaining 3 bits of PLI_4. Specifically, the adversary guesses the low-order 3 bits of PLI_4, creates a mapping based on this guess, and checks if the secret-dependent page walk results in an additional cache fill (due to PLI_3's access). If an additional cache fill occurs, the adversary learns both the low-order 3 bits of PLI_4 and the high-order 6 bits of PLI_3. The adversary can repeat this process, visualized in Figure 5, one PT level at a time and, at best, reconstruct the entire VPN bits (47:12) and half of the page offset bits (11:6)—a total of 42 bits—of `secret`.

The memory mapping order oracle has one limitation. If `secret` shares any subset of its PT indexes (from PLI_4 to PLI_i for $1 \leq i \leq 4$) with memory mappings that already exist in the victim's virtual address space, their respective PTs will always be present and the secret-dependent page walk will never fail at those PT levels. As a result, the cache fills associated with those PT accesses will always occur. Fortunately, in Sections 4 and 5 we find this situation rarely happens. Nonetheless, we now present an alternative order oracle method that can be used when this happens.

Page walker cache order oracle. The second order oracle method involves selectively flushing different page walker caches to cause the secret-dependent page walk to only access specific, attacker-selected PT levels. Recall from Section 2.1 that before starting a page walk, the MMU checks if the various page walker caches (starting from the 1st level one, also known as the TLB) contain a full or partial translation of `secret`'s VPN bits. If there is a cache hit, the MMU uses the cached translation to skip as many PT

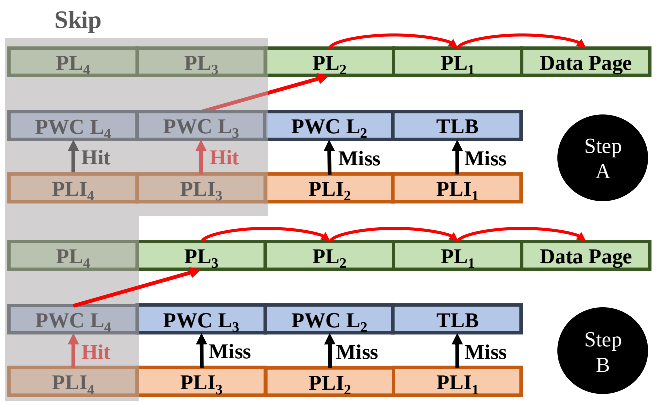

Figure 6. By manipulating the page walker caches (PWCs) an attacker can hide portions of the page walk process and conduct a sliding window style attack to order the page walk accesses. For example, when the attacker flushes the TLB and PWC L_2 (top), accesses to PL_2, PL_1 and data page are revealed. If the attacker further flushes the PWC L_3 the attacker will observe an extra access compared to just flushing the PWCs up to L_2. The extra access reveals to the attacker PLI_3.

accesses as possible. For example, if there is a hit in the 3rd level page walker cache, the MMU skips accessing PL_4 and PL_3 and directly moves to the PL_2 access.

Using the page walker cache order oracle, the adversary progressively increases the levels of the page walker caches that are selectively flushed (starting from the 1st level), one at a time.[7] This way, only one new PT index is encoded into the cache state at each round. By proceeding one level at a time, this order oracle directly maps each cache set index to its corresponding PT level or data page access. Figure 6 visualizes this process, which, when `secret` shares its entire VPN with that of pre-existing memory mappings, allows the adversary to reconstruct the high-order 6 bits of each PT index and the high-order 6 bits of the page offset—a total of 30 bits—of `secret`.

The page walker cache order oracle can be used to overcome the aforementioned limitation of the memory mapping order oracle at the cost of a slightly reduced bit leakage capability (i.e., the low-order 3 bits of the already-mapped PT indexes cannot be reconstructed). Further, the page walker cache order oracle can be used in threat models where `secret` is not allowed to map to the attacker's virtual address space. In these threat models, however, the oracle is limited to leaking the subset of PT indexes that is shared with pre-existing memory mappings. If no such mappings exist and the adversary cannot create them, this oracle can only leak the high-order 6 bits of PLI_4.

3.3. PWSC summary

Overall, given an arbitrary 64-bit value `secret`, PWSC involves a transmitter executing `*secret` in the victim's security domain and a receiver that attempts to reconstruct

7. We describe our approach to selectively flush the various page walker cache levels in Appendix A.

```
    // Architecturally
    nested_addr = *addr
    ... = *nested_addr

    // Under Speculation
    // Attacker Controls addr
    secret = *addr
    ... = *secret
```

Listing 1. Pointer chasing pattern that can become a `*secret`.

`secret` from the attacker's security domain. PWSC's receiver leaks the bits of `secret` one PT index at a time. Specifically, the receiver uses differential Prime+Probe (Section 3.1) and an order oracle (Section 3.2) to leak the high-order 6 bits of each level's PT index and of the page offset. When using the memory mapping order oracle, the receiver additionally leaks the low-order 3 bits of each level's PT index. At best, PWSC is capable of leaking 42 bits of `secret` (i.e., bits 47:6), which exceeds the leakage capability of all previously known memory-based side channels. However, PWSC makes a fundamental assumption whose applicability we have yet to discuss: that the receiver can cause `*secret` to induce a page walk. In most systems, this assumption requires passing a canonicality check, meaning that bits 63-48 of `secret` must be copies of the 47th bit. We revisit this limitation more in detail in the following sections.

4. Case study 1: data dereferencing through speculation

In this section, we demonstrate that PWSC increases information leakage relative to the state of the art when secret data is dereferenced as a result of speculative execution. Section 4.1 provides background and the threat model. Section 4.2 describes our approach for combining Spectre-BHI and PWSC. Section 4.3 demonstrates that, when the Intel LAM processor feature is enabled, PWSC in conjunction with unmasked Spectre gadgets can be exploited to dump almost all of physical memory. Finally, Section 4.4 demonstrates that, even without LAM, we can combine PWSC with unmasked Spectre gadgets to perform full key extraction on the post-quantum Dilithium cryptosystem.

4.1. Preliminaries

Threat model. We assume the standard user-to-kernel Spectre threat model. In this setting, the adversary has full control of an unprivileged user process and aims to leak secret data from the kernel. We assume all currently deployed kernel mitigations to Spectre attacks are enabled. Additionally, when exploiting unmasked gadgets, we assume that the adversary knows the address of `secret`. This assumption follows prior work, which describes ways to leak secret's address either by using a leakage gadget to scan the kernel or by massaging physical memory [14], [48], [49].

Unmasked gadgets. Recent work [14] reports that the majority (numbering in the tens of thousands) of candidate Spectre gadgets in the Linux kernel follows the pointer-chasing pattern shown in Listing 1, where the code performs two nested memory dereferences. Architecturally, these gadgets are harmless, as they always dereference non-secret, valid pointers. However, under speculation, an adversary who controls the input address (`addr`) can cause these gadgets to dereference (transmit) secret, non-pointer data.

These gadgets are known as *unmasked gadgets* as they do not apply any mask to the secret before transmission [16]. Unmasked gadgets are difficult to exploit for several reasons. One important reason is that, for `*secret` to initiate a memory dereference, `secret` needs to pass a canonicality check (Section 2.1). Hertogh et al. [14] circumvented this by abusing a feature of upcoming Intel processors called Linear Address Masking (LAM). They then used an Evict+Reload covert channel through the data TLB (dTLB) to extract data.

Linear address masking. LAM is a feature of upcoming Intel microarchitectures (e.g., Sierra Forest, Grand Ridge, Arrow Lake, and Lunar Lake) that allows software to use the most significant bits of virtual addresses for metadata [14], [50]. To this end, LAM relaxes the canonicality check on the lower 15 bits of the most significant 16 sign-extended bits in a virtual address and can be enabled for either all or only userspace addresses (in which case, we call it *userspace LAM*). When LAM is enabled, the canonicality check just verifies whether the 63rd and 47th bits of the virtual address are equal, rather than checking equality of all 63:47 bits. In our experiments, we simulate LAM in software by sign-extending the 47th bit of addresses to bits 62:48, as done by Hertogh et al. [14].

State-of-the-art receiver for unmasked gadgets. Hertogh et al. [14] use an Evict+Reload dTLB receiver to leak data from unmasked gadgets. Their receiver consists of (i) mapping a large reload buffer, (ii) evicting the dTLB entries of each item in said buffer, (iii) using an unmasked gadget to dereference secret data whose VPN is shared with that of a valid address in the reload buffer and (iv) measuring the time to complete dereferences of addresses in the reload buffer to find which result in TLB hits. Unfortunately, this features an important limitation: it requires that the secret data have low entropy. This is because a high-entropy secret would require prohibitively large reload buffers that exceed the size of the TLB and become practically unmanageable for an attacker. Due to this restriction, Hertogh et al. [14] were only able to leak kernel ASCII strings with a known prefix, i.e., those found in `/etc/shadow`.

4.2. Exploiting unmasked gadgets with PWSC

We now present our methodology for increasing the amount of information leakable by an adversary using unmasked gadgets. Like Hertogh et al. [14], we rely on the Spectre Branch History Injection (BHI) primitive to trigger mis-speculation. Unlike Hertogh et al. [14], we rely on the novel PWSC channel to transmit and receive secrets. We now summarize how we combined these two components.

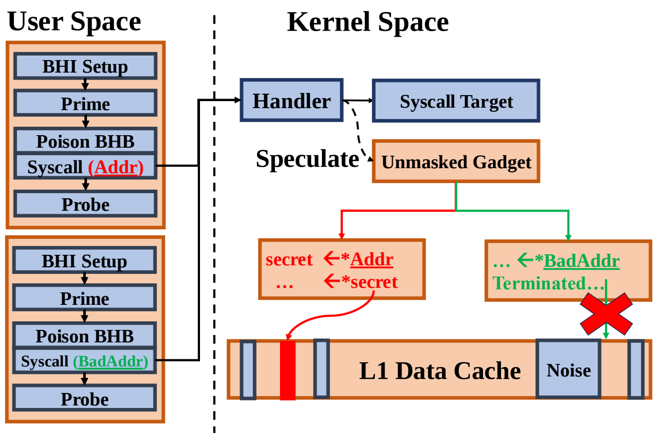

Figure 7. The final Spectre-BHI + PWSC procedure. The attacker wraps the BHB poisoning and the victim system call in the differential Prime+Probe measurement. By passing in a noncanonical address (BadAddr) the adversary can selectively disable the execution of `*secret`.

Spectre BHI. Spectre BHI [29] is a cross-privilege Spectre-V2 attack where an adversary poisons the branch history buffer (BHB) to trigger mis-speculation. When encountering an indirect branch, the CPU speculatively jumps to the code target predicted by the branch target buffer (BTB). The BTB is indexed by both the current instruction address and the conditional branch history stored in the BHB. The BHB stores the outcomes of the last N (on our processor, $N = 194$) dynamic conditional branches. By poisoning the BHB, the adversary can direct where the processor looks in the BTB and redirect execution to a gadget of their choosing.

In all our experiments, we target the 10 BHI-compliant unmasked gadgets exploited by Hertogh et al. [14][8] and use the same steps to launch Spectre-BHI as prior work [14], [29], [38]. Specifically, the adversary (i) primes the BTB, (ii) increases the speculation window so that the unmasked gadget executes speculatively, (iii) sets the registers used as inputs to the target gadget to the desired values, (iv) poisons the BHB, and (v) makes a system call into the kernel which triggers mis-speculation to the desired gadget.

Combining Spectre-BHI and PWSC. To launch Spectre-BHI with the PWSC receiver, the adversary first preconditions system state for the order oracle. The exact details of this process depend on which level of the page walk we are analyzing and which order oracle is available (Section 3.2). Second, the adversary performs both the above-described Spectre-BHI steps and the differential Prime+Probe steps from Section 3.1 to trigger a page walk on `secret` and monitor the associated cache fills. To avoid polluting the BHB, we perform the Prime step immediately before poisoning the BHB. Finally, to selectively disable the execution

8. Currently deployed Spectre mitigations do not protect against these unmasked gadgets, presumably due to their perceived exploitation difficulty.

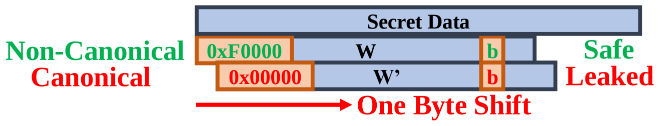

Figure 8. By sliding the attacker's window, the attacker has several chances to leak a given bit b.

of `*secret` (as required by the differential Prime+Probe receiver), the adversary picks an address `addr` that fails the (relaxed) canonicality check. These attack steps are summarized in Figure 7.

PWSC order oracle choice. Suppose we attempt to use the above procedure to leak a 64-bit secret value W that passes the (relaxed) canonicality check. The MMU will interpret W as either a userspace or kernelspace address depending on the kernel address bit (Section 2.1). If W looks like a userspace address, PWSC can leverage the memory mapping order oracle to leak, with overwhelming probability, its entire VPN bits (bits 47:12).[9] If SMAP is disabled, PWSC can additionally leak the high-order 6 bits of W's page offset (bits 11:6). If W looks like a kernel-space address, however, PWSC must rely on the page walker cache order oracle and can only leak between 6 and 30[10] bits of W, depending on whether pre-existing kernel memory mappings share their VPN bits with W (Section 3.2).

4.3. Attacking systems with LAM

We now show that Spectre-BHI and umasked gadgets, combined with PWSC, can be used to leak almost all of physical memory (via physmap, the Linux kernel's memory map to the contents of physical memory) when LAM is enabled. The attack makes no assumptions on the contents of physmap, i.e., it does not assume any known prefix in or assume certain distributions of the secret data. In particular, the secret data does not need to represent valid pointers.

Exploiting LAM. Recall from Section 4.1 that LAM disables the canonicality check on the lower 15 bits of the upper 16 sign-extended bits in a virtual address. If we assume physmap is full of random-looking data, this suggests that $\sim 50\%$ of physmap will be unleakable using just PWSC. Further, of the data that does pass the relaxed canonicality check, some will be out of reach of PWSC's receiver (e.g., the cache line offset bits in the virtual address page offset).

We circumvent the above issues by using a sliding window strategy that slides at byte granularity. Figure 8 gives an example. Suppose we wish to leak bit b contained in a 64-bit word W that fails the relaxed canonicality check. By sliding one byte, we now attempt to dereference the

9. Observe that in this situation the adversary controls all the available userspace memory mappings since they all fall within their own address space. The adversary can thus minimize pre-existing memory allocations to ensure (with overwhelming probability) that the secret-dependent page fault will fail at attacker-selected levels.

10. SMAP does not apply to kernel addresses.

TABLE 1. AVERAGE ACROSS PHYSICAL MEMORY PROFILING.

	LAM	Userspace LAM Only
Total	99.55%	95.99%
Interesting	98.13%	81.02%

shifted word W' that still contains b. Although W failed the check, W' may pass the check in which case we have another chance to learn b. The same principle allows us to leak additional bits, e.g., the page offset bits. The only way to fail to leak a bit b is for all slides in the vicinity of b to yield words that fail the relaxed canonicality check.

Physmap leakage analysis. We now assess what bits of physmap can be leaked with an unmasked gadget combined with our PWSC receiver and the above sliding window procedure. To this end, we use LiME [51] to extract the data currently in physmap (16 GB) three times with different applications running. We then collect statistics on that data.

For our analysis we conservatively assume that no memory is mapped, i.e., all page walks fail at PT level 4. We also assume that SMAP is enabled. This means that for secrets that appear as userspace pointers, the adversary can leak all VPN bits, and for secrets that appear as kernel-space pointers, they can only leak the high-order 6 bits of PLI_4.

We find that there are many all-zero bytes (0x00) in the physical memory. Since all-zero bytes pass the canonicality check but are often unused memory or the result of zeroing out memory (i.e., not interesting to leak), we present two metrics. The "Total" metric counts the percentage of bits that are leakable. The "Interesting" metric counts the percentage of bits that are leakable that are not a part of an all-zero byte. We do not check for bits that are leaked as a part of an all-one byte (0xFF) but note that this case was uncommon.

Table 1 reports the results. If LAM is enabled, almost all memory ($> 99\%$) is leakable with unmasked gadgets and PWSC's receiver. If only userspace LAM is enabled, this number ($> 96\%$) is still almost all memory. These numbers are close to 100% because of the sliding window strategy discussed earlier: by shifting byte by byte the adversary has multiple chances to leak a given byte, and only one of the windows containing the byte needs to look like a canonical pointer for the attack to work. Next, we evaluate the attack in practice against both ASCII and random kernel data.

Leaking ASCII data. To start, we select two ASCII targets: `/etc/shadow` and arbitrary ASCII strings owned by a victim process. The first target is vulnerable to known attacks [14] and we report it here for completeness. The second target is only leakable using PWSC, as arbitrary strings in a victim process do not have a known prefix. For the second target, we insert a 700-character random string in a victim process's heap, text, and stack segments.

Recall that ASCII strings are encoded in 7 bits and leave the most significant bit as 0. Without LAM, any 64-bit word in an ASCII string would fail the canonicality check, preventing the attack. With LAM, however, any such word

TABLE 2. COMPARISON OF SLAM ACCURACY VS. PWSC ACCURACY ACROSS ASCII AND NON-ASCII SECRETS.

	SLAM	PWSC
ASCII	>99%	>99%
Non-ASCII	Unable to leak bits	81.95%

will appear as a canonical LAM userspace pointer [14]. The adversary can therefore use the memory mapping order oracle to extract the VPN bits of any such word. Further, since we know that every ASCII character will be canonical we can also improve performance by not using a sliding window strategy and just leaking each character one-by-one.

We now report the results of the attack. For the `/etc/shadow` experiment, after feeding in the target kernel address, the attack can extract the root password hash in 68 seconds. For the random ASCII victim process experiment, the attack can extract the target string from the victim process at a rate of 0.88 chars (6.16 bits) per second. Sometimes, extra system noise in the differential step of PWSC can result in misreporting certain characters. Nonetheless, the attack succeeds with $> 99\%$ accuracy.

Leaking non-ASCII data. Finally, we evaluate the attack on random portions of physmap that contain "Interesting" kernel data for a total of 19064 bits. We find that PWSC's receiver can extract this memory at a rate of 3.064 bits per second and extracts 81.95% of the bits correctly. Some bits are not reconstructed correctly due to extra noise subtracted from the signal in PWSC's differential Prime+Probe receiver. Moreover, the bit rate is lower compared to ASCII leakage due to the sliding window strategy causing the same bits to be leaked multiple times, slowing down leakage. We compare PWSC's accuracy to SLAM's accuracy in Table 2.

4.4. Attacking systems without LAM

We now demonstrate that Spectre-BHI and umasked gadgets, combined with PWSC, can be used to leak sensitive data even on systems where LAM is not available or disabled. Specifically, we perform a full key extraction attack on the Dilithium post-quantum cryptosystem.

Side-stepping canonicality checks. Without LAM, the values recoverable using PWSC are severely restricted. Nonetheless, PWSC can still be used to leak data that appears canonical. Specifically, the adversary can use PWSC to leak the bits that follow a pattern of 17 consecutive 0 bits (0x00000) or 17 consecutive 1 bits (0xFFFF8).[11] A pattern of 17 consecutive 0s means that the data looks like a userspace pointer; given this pattern, the adversary can use the memory mapping oracle to leak, with overwhelming probability, all the following 35 bits (the 'VPN' bits).[12] A pattern of 17 consecutive 1s means that the data looks like a kernel-space pointer; given this pattern, the adversary can use the page walker cache order oracle to leak between 5 and 29 of the bits that follow (cf. Section 4.2).

We now demonstrate how, without LAM, the above principle can be exploited to extract the full cryptographic key from the Dilithium post-quantum cryptosystem.

Background. Dilithium, also known as ML-DSA, is a digital signature scheme approved by NIST's post-quantum cryptography project [52]. Dilithium is based on the module learning with errors (MLWE) and the module short integer solution (MSIS) problems. Its computations are done in the polynomial ring R_q ($\mathbb{Z}_q[x]/x^n + 1$), i.e., the ring of polynomials with coefficients in $\mathbb{Z}_q$ and modulo $x^n + 1$. Let R_q^k denote a vector of k elements where each element is in R_q. The Dilithium procedure starts by generating a public and secret key pair. Specifically, $\mathbf{s}_1$ ($\mathbf{s}_1 \in R_q^l$) is one of the secret key components. Coefficients in $\mathbf{s}_1$ are uniformly distributed in the range $[-\eta, \eta]$. The goal of our attack is to leak $\mathbf{s}_1$ as it gives the attacker the ability to forge signatures.

Dilithium's signing procedure samples the nonce $\mathbf{y}$ ($\mathbf{y} \in R_q^l$) deterministically (based on the secret and message) or non-deterministically (for every signature). Coefficients of $\mathbf{y}$ are distributed uniformly in $[-\gamma_1, \gamma_1]$. The algorithm uses the message, secret key and nonce to compute the signature that follows the constraint that $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$, where $\mathbf{z}$ ($\mathbf{z} \in R_q^l$) and c ($c \in R_q$) are components in the output signature.

Leaking Dilithium keys. We consider victim running the reference, side-channel hardened implementation of Dilithium-2 [52] in a userspace process on the same machine as the attacker. The victim generates a Dilithium key pair and signs incoming messages. Every time the victim signs a message (e.g., in response to attacker-provided requests), the corresponding nonce $\mathbf{y}$ is produced and stored in the stack. The attacker can then locate this nonce $\mathbf{y}$ in physmap and use PWSC to recover its coefficients. The leaked coefficients serve as hints that can be passed to lattice reduction tools to fully reconstruct the secret key component $\mathbf{s}_1$ [53].

In Dilithium-2, a nonce $\mathbf{y}$ is a vector of four 256-degree polynomials ($n = 256$, $l = 4$). Each coefficient of $\mathbf{y}$ is a 32-bit integer ranging from -2^{17} to 2^{17} ($\gamma_1 = 2^{17}$). Represented in binary, one coefficient could range from 0x00000000 to 0x00020000 (positive) or from 0xFFFFFFFF to 0xFFFE0000 (negative). Two consecutive coefficients form a 64-bit address. Thus, the probability that an address looks like a canonical pointer is 25% (or 12.5% if we only count userspace pointers). With our PWSC receiver, the adversary can attempt to leak one coefficient at a time. We apply a 32-bit sliding window so that the most-significant bits of the value triggering the page walk are always the most-significant bits of a 32-bit coefficient.

Each leaked 32-bit coefficient can be used to construct a linear equation that encodes information about $\mathbf{s}_1$. Given a nonce $\mathbf{y}$, it holds that $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$. Here, $\mathbf{z}$ and c are exposed to the attacker through the output signature. Suppose the first coefficient of the nonce is leaked ($\mathbf{y}[0][0]$). The attacker can

11. Since x86 is little endian, we say that a byte at address x follows bytes at addresses $x + 1$.

12. The most significant bit of the VPN is included in the canonicality check. Therefore, the leakable bits are the VPN bits minus that one bit.

construct the equation as $c\mathbf{s}_1[0][0] = \mathbf{z}[0][0] - \mathbf{y}[0][0]$ and pass it to a lattice reduction tool. In [53], full key recovery requires 876 hints, which means that our attack needs the victim to sign approximately 4 messages or 7 messages if the attacker can only leak userspace addresses. In practice, we find that it takes an adversary 115 minutes to collect 921 hints for the lattice reduction. The lattice reduction then takes 81 minutes to successfully extract the secret key.

5. Case study 2: data dereferencing through data-memory dependent prefetchers

Recent studies have found that data memory-dependent prefetchers (DMPs) on modern Apple CPUs scan and dereference pointer-like data directly from the memory system [15], [42]. DMPs, like speculative execution, also effectively suppress the exception that would result from dereferencing an invalid pointer. That is, the DMP can microarchitecturally dereference `secret` (`*secret`), creating a second path through which to create an unmasked gadget (cf. Section 4.1). However, prior attacks only leak limited information about the secret dereferenced by the DMP (i.e., if it looks like an attacker-chosen pointer [15]) and fail when the secret is an invalid pointer, as they rely on conventional cache side-channel receivers.

In this section, we combine the DMP memory access with PWSC to extend the leakage capabilities of DMP attacks and demonstrate conclusively that DMPs can be used to leak invalid pointers. To this end, we reverse engineer the precise activation criteria of the DMP on our Intel i9-13900K processor (Section 5.1). We then show how the Intel DMP and PWSC can be combined to break security in an intra-process sandbox setting (Section 5.2).

5.1. Intel DMP characterization

We begin by reviewing the basic DMP activation pattern described in Augury [42] and GoFetch [15]. This pattern first initializes an array (`aop`) of length M and fills `aop` with pointers to memory addresses that correspond to unique cache lines. Next, it uses `clflush` to evict both `aop` and the memory addresses pointed to by `aop`. The code then architecturally dereferences the first N entries in `aop` (with $N < M$), as illustrated in Figure 9 (first row) and Listing 2. Finally, the code uses `rdtscp` to time the latency of dereferencing "out-of-bounds" `aop` pointers that had not previously been dereferenced, i.e., `aop[N],...,aop[M-1]`.

With $N \geq 300$, we observe consistent L1 cache hits ($\sim$ 6 cycles) when dereferencing 16 out-of-bounds pointers (i.e., entries $[N, N+15]$ of `aop`). We find that for the DMP to activate, `aop` must be 64-bit (size of a pointer) aligned. We also tested the other (Apple) DMP activation patterns described in GoFetch [15] but found that only the above activation pattern activates the Intel DMP.

Preconditions for activation. We split the previous access pattern into two phases: (i) a training phase, used to train the Intel DMP to dereference out-of-bounds pointers and (ii) a prefetching phase, used to activate a pre-trained DMP.

```
// Fill aop with unique pointers
uint64_t aop[M];
for(int i=0; i<N; i++) {
    ptr = aop[i];
    ... = *ptr;
}
```

Listing 2. The access pattern studied by GoFetch that activates Intel DMPs.

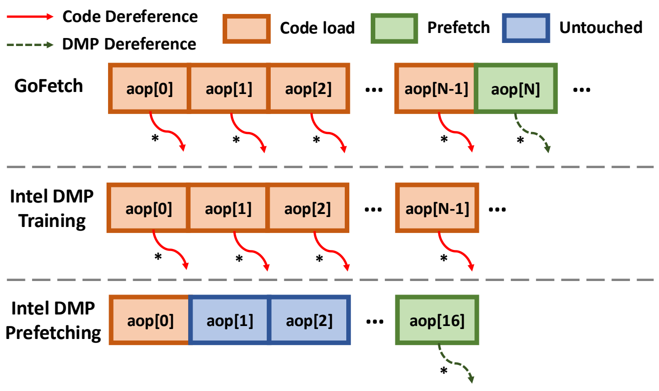

Figure 9. The first row shows the memory access pattern used by GoFetch to activate the Intel DMP [15]: a streaming access pattern that architecturally loads and dereferences pointers. We split this access pattern into two phases: training and prefetching (second and third row). The training phase matches the original access pattern, while the prefetching phase has less restrictions and can trigger the DMP even with just one memory load.

Using the same access pattern as before for the training phase (second row of Figure 9), we aim to test if the DMP (i) is *PC tagged*, meaning it only activates on memory loads with the same PC (program counter) as the loads used for training, and (ii) only activates when the memory accesses originate from the same `aop` used for training.

We perform two experiments. In the first one, the prefetching phase reuses the same instructions as the training phase but streams over a different `aop`. In the second one, the prefetching phase accesses the same `aop` as the training phase but executes a separate copy of the training code. Both experiments train the DMP with $N = 300$. We only observe DMP activations (i.e., dereferences of out-of-bounds pointers) in the first experiment, suggesting that the Intel DMP is PC tagged and the prefetching phase does not need to rely on the same `aop` used for the training phase.

Avoiding architectural pointer dereferencing. Next, we investigate which of the two load instructions of Listing 2 (at Lines 4 and 5) is used for the PC tag. To this end, we run a variant of the prefetching phase where only one of the two load instructions is executed. To ensure that both phases still use the same code, we insert an "if" statement (with barriers to inhibit speculative execution) to conditionally skip one of loads during the prefetching phase. We observe that performing the first load (Line 4) is sufficient to activate the DMP. We conclude that the DMP is PC tagged to the *first load* of the training pattern and that architectural pointer dereferences are not required to activate the DMP during the prefetching phase. We refer this load as the *DMP trigger*.

As a byproduct of this experiment, we find that activating the DMP with only the first load decreases its confidence. That is, when we stream over `aop` without dereferencing pointers, we eventually stop observing DMP activations. Specifically, in our experiments, we stop seeing DMP activations after executing 10 DMP triggers.

PC aliasing. It is common for microarchitectural tags to only track partial bits to reduce overheads [54]. To test if the DMP PC tag is vulnerable to aliasing attacks, we modify the prefetch phase to use a DMP trigger with a 1-bit PC tag difference compared to the trained PC tag. We move the 1 bit difference from the least significant to the most significant bit until we observe DMP activations. This would imply that the two PC tags are aliased, revealing the number of bits used for the PC tag. Our experiment shows that the Intel DMP only stores the least significant 10 bits of the training load's PC. That is, the DMP activates when the least significant 10 bits of a load's PC match the least significant 10 bits of the training load's PC.

Dereference target. Next, we aim to investigate *what* addresses the Intel DMP dereferences when it activates. To start, using the same training pattern as above, we execute a DMP trigger loading from `aop[j]`, where `aop` is a different array of pointers than the one used for training. We then time the latency of dereferencing entries $[\texttt{j}+1, \texttt{j}+16]$ of `aop`. We observe that, when accessing `aop[j]`, the DMP dereferences the address stored at `aop[j+16]` (i.e., 128 bytes ahead of the address loaded by the DMP trigger).

We also check if the pattern of loads during training influences the Intel DMP behavior during the prefetching phase. To this end, we modify the stride used to access the array of pointers during training[13] and check if the stride influences what address the DMP dereferences when it activates. This time, we find that, when accessing `aop[j]`, the DMP dereferences the address stored at $\texttt{aop[j} + \texttt{stride} \times 16\texttt{]}$. That is, the DMP dereferences pointers stored $\texttt{stride} \times 16$ entries (i.e., $\texttt{stride} \times 128$ bytes) ahead of the address loaded by the DMP trigger.

We summarize the DMP activation function in Figure 9 (third row). This function implies that the DMP can dereference data that was and will never be architecturally accessed by the program. Further, the attacker can *precisely target* an address for the DMP to dereference microarchitecturally.

DMP restrictions. Finally, we investigate restrictions on the Intel DMP across three dimensions: (i) training the DMP across process boundaries; (ii) restrictions on pointers the DMP will attempt to dereference; (iii) maximum prefetch distance from the DMP trigger load. For (i), our results show that the DMP cannot be trained across process boundaries, both when time-sharing the same logical core and in an SMT setting, matching Intel's documentation [47]. For (ii), we find that the DMP will attempt to dereference canonical userspace pointers even if they are invalid addresses but will not attempt to dereference kernel space or noncanonical pointers. For (iii), we find that the address accessed by the DMP trigger must live in the same 256 KB aligned region as the address to be dereferenced for the DMP to activate.

5.2. Attacking Intra-Process Sandboxes

We now demonstrate how an attacker can use PWSC (Section 3) and the DMP (Section 5.1) to leak data outside its security domain in an intra-process sandbox setting.

Threat model. We assume an intra-process sandbox scenario. Each security domain is restricted to accessing data within its own address space, enforced by memory permission management primitives. Specifically, we use ERIM [55], an Intel MPK-based intra-process sandbox for our attacks. We assume the victim exposes a function call interface to the attacker, and that this call is protected against Spectre attacks through speculation barriers. The attacker's goal is to leak secrets from the victim's address space. We assume that the target secrets pass the address canonicality check (either due to LAM or due to starting with 17 consecutive 0 bits or 1 bits, as discussed in Section 4).

Attack overview. First, we attempt to use the DMP to dereference the victim's secrets from the attacker's security domain. However, we observe that the Intel DMP follows the memory permission rules enforced by MPK and cannot directly dereference data across intra-process sandboxes.

We circumvent the above limitation by exploiting the PC aliasing behavior of the DMP and causing the DMP to activate from the victim's security domain. Suppose the attacker wants to leak a secret stored at address `addr` (in the victim's security domain) and that `addr` is 64-bit aligned. The attacker first identifies a load in the victim's function accessing an address `victim_addr`, where `victim_addr` is a multiple of 128 bytes away from and lives in the same 256 KB region as `addr`.[14] Next, the attacker computes a stride $\texttt{s} = (\texttt{addr} - \texttt{victim\_addr})/128$. The attacker then runs a training loop in their own address space, using a stride `s` and with a load instruction that PC aliases with the victim load of `victim_addr`. After training, the attacker can trigger a secret-dependent page walk by calling the victim function. Specifically, when the victim function loads from `victim_addr`, the DMP will load and dereference the pointer stored at $\texttt{victim\_addr} + \texttt{s} * 128 = \texttt{addr}$.

Evaluation. We evaluate our attack on 1,000 randomly generated values that are invalid pointers. We force these values to be canonical by fixing the most significant 17 bits to be zeros, leaving 42,000 bits (bits 47:6 of each value) for

13. Line 4 in Listing 2 can be replaced with `ptr = aop[i*stride]` to enable various training strides.

14. In some cases, the attacker may have control over `victim_addr` via passing arguments to the victim's function.

PWSC to leak. Our implementation can leak these bits at a rate of 14.46 bits per second and an accuracy of 82.08%. Similarly to Section 4, some bits are not reconstructed properly due to noise.

6. Mitigations

For case study 1, our attacks could be mitigated using prior Spectre defenses (e.g., eIBRS [22]) and LASS, an upcoming Intel feature that enforces kernel-userspace address space isolation [22], [50]. Specifically, LASS prevents kernel code from initiating page walks on userspace addresses, which would prevent Section 4's attacks from leaking data that looks like userspace pointers. However, even with LASS enabled, an attacker could still use the PWC order oracle to leak between 6 and 30 bits of data that looks like canonical kernelspace pointers, as discussed in Section 4.2.

For case study 2, our attacks could be mitigated using DMP-specific mitigations, which include disabling the DMP (with Intel DOIT) and/or using the IBPB command when switching between different security domains [47]. At the time of writing, however, neither of these commands is accessible to unprivileged software.

7. Related Work

Recovering sensitive information through page walks. AnC also leverages the cache side effects of page walks to leak secret PT indexes [17]. However, as we discuss in Section 2.3, their assumptions and setting are completely different from ours. Specifically, AnC assumes write access to the public target pointer, whereas we assume no write access to the secret target pointer. AnC's assumptions dramatically simplify their attack, as the attacker can easily isolate target page walk levels, does not have to cope with invalid pointers (as we do), and deals with significantly less noise—obviating the need for our mechanisms in Section 3. Correspondingly, AnC can only be used to break ASLR, whereas PWSC can be used as a general transmission primitive for microarchitecture that attempts to dereference secrets (e.g., speculative execution, DMPs).

Binoculars [56] exploits a different mechanism (false dependencies between loads and stores), to infer the VPN. However, they do not purpose a general order oracle and assume external knowledge (e.g., that the secret is a stack address) or that the secret is otherwise low entropy (e.g., limited to attacking KASLR). Additionally, neither AnC nor Binoculars consider leaking invalid pointers.

Exploiting unmasked gadgets using speculative execution. Hertogh et al. [14] use dTLB side channels and LAM to exploit unmasked gadgets due to speculative instruction execution. Due to their use of a dTLB receiver, they are restricted to just a small subset of the kernel's direct memory region (`/etc/shadow`) or, more generally, ASCII strings with a known prefix. Moreover, they do not explore attacks without LAM enabled. Our work shows that the consequences of LAM are significantly worse than shown in SLAM—namely that PWSC renders almost all of physical memory vulnerable. We also explore attacks without LAM enabled and show that even without LAM, valuable data is still vulnerable. Furthermore, we show that unmasked gadgets can form due to microarchitectural optimizations beyond speculative instruction execution (such as the DMP).

Apple DMP. Augury [42] and GoFetch [15] study the DMP in Apple silicon. While Augury does not give a realistic attack target, GoFetch performs cryptanalysis to coerce key-dependent data to "look like" a pointer. We mainly view our work as orthogonal to these: our focus is to provide a generic mechanism to increase leakage through dereferenced invalid pointers through the PWSC procedure. GoFetch also confirmed the existence of the Intel DMP and determined a basic activation pattern for it. However, our work is the first to reverse engineer the semantics of this DMP.

Since we combine PWSC with the Intel DMP, a natural question to ask is whether our techniques extend to the Apple DMP. We believe the answer is primarily "no." The Apple DMP does not have a training phase and attempts to dereference all eight pointers within the incident loaded cache line. This makes denoising more challenging. Further, Apple DMPs refuse to dereference data whose most significant 32 bits do not match the data's address, which precludes many bits from being leaked.

8. Conclusion

Cache side-channels are by now a well-known and well-understood tool for attackers. Unsurprisingly, the best-practices for secure code have long included disallowing secret-dependent memory accesses of any kind under the assumption that the secret will leak. While conceptually true, practical attacks have always required workarounds and additional gadgets for manipulating the secret bits into the portions of the address that have known techniques for receiving them. Worse for the attacker, when a non-pointer secret flows to a gadget measured by a cache side-channel, no leakage occurs. This has not gone unnoticed by defensive proposals and software maintainers, who have avoided removing Spectre gadgets when no realistic receiver is known. Our work makes significant progress towards aligning the implicit threat model in best practices and actual attacker's capabilities. If we assume a standard cache and TLB based system, the page walk side channel presented leaks *all* of the bits encoded into microarchitectural state by a secret-dependent page walk.

Acknowledgment

This work was partially supported by NSF grants CNS-1954521, CNS-1942888, CNS-2154183, CNS-2153388, the Air Force Office of Scientific Research (AFOSR) under award number FA9550-24-1-0079, the Defense Advanced Research Projects Agency (DARPA) under contract numbers W912CG-23-C-0022, an Alfred P Sloan Research Fellowship, and gifts from Qualcomm, Cisco, and Intel.

References

[1] C. Percival, “Cache missing for fun and profit,” in *BSDCan*, 2005.

[2] D. A. Osvik, A. Shamir, and E. Tromer, “Cache attacks and countermeasures: The case of aes,” in *CT-RSA*, 2006.

[3] Y. Yarom and K. Falkner, “Flush+reload: A high resolution, low noise, l3 cache side-channel attack,” in *USENIX Security*, 2014.

[4] D. Gruss, R. Spreitzer, and S. Mangard, “Cache template attacks: Automating attacks on inclusive last-level caches,” in *USENIX Security*, 2015.

[5] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, “Last-level cache side-channel attacks are practical,” in *IEEE S&P*, 2015.

[6] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, “Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX,” in *USENIX Security*, 2017.

[7] D. Genkin, L. Valenta, and Y. Yarom, “May the fourth be with you: A microarchitectural side channel attack on several real-world applications of Curve25519,” in *CCS*, 2017.

[8] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware,” *JCEN*, 2018.

[9] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri, “Port contention for fun and profit,” in *IEEE S&P*, 2019.

[10] Y. Yarom, D. Genkin, and N. Heninger, “CacheBleed: A timing attack on OpenSSL constant time RSA,” *JCEN*, 2017.

[11] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi, “ABSynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures,” in *NDSS*, 2020.

[12] R. Paccagnella, L. Luo, and C. W. Fletcher, “Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical,” in *USENIX Security*, 2021.

[13] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan, “Don’t mesh around: Side channel attacks and mitigations on mesh interconnects,” in *USENIX Security*, 2022.

[14] M. Hertogh, S. Wiebing, and C. Giuffrida, “Leaky address masking: Exploiting unmasked spectre gadgets with noncanonical address translation,” in *S&P*, 2024.

[15] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, “GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers,” in *USENIX Security*, 2024.

[16] Intel, “Intel research on disclosure gadgets at indirect branch targets in the linux* kernel,” https://www.intel.com/content/www/us/en/developer/articles/news/update-to-research-on-disclosure-gadgets-in-linux.html, accessed on Jun 6 2024.

[17] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, “ASLR on the line: Practical cache attacks on the MMU,” in *NDSS*, 2017.

[18] M. Neve and J.-P. Seifert, “Advances on access-driven cache attacks on AES,” in *SAC*, 2006.

[19] O. Aciiçmez, “Yet another microarchitectural attack: Exploiting i-cache,” in *CSAW*, 2007.

[20] O. Acıiçmez and W. Schindler, “A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL,” in *CT-RSA*, 2008.

[21] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM side channels and their use to extract private keys,” in *CCS*, 2012.

[22] Intel, “Hardware features and behavior related to speculative execution,” https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/hardware-behavior-related-to-speculative-execution.html, accessed on Oct 11 2024.

[23] ——, *Intel 64 and IA-32 Architectures Software Developer’s Manual*, April 2024.

[24] T. W. Barr, A. L. Cox, and S. Rixner, “Translation caching: skip, don’t walk (the page table),” in *ISCA*, 2010.

[25] A. Bhattacharjee, “Large-reach memory management unit caches,” in *MICRO*, 2013.

[26] C. H. Park, I. Vougioukas, A. Sandberg, and D. Black-Schaffer, “Every walk’s a hit: making page walks single-access cache hits,” in *ASPLOS*, 2022.

[27] S. Van Schaik, K. Razavi, B. Gras, H. Bos, and C. Giuffrida, “RevAnC: A framework for reverse engineering hardware page table caches,” in *EuroSec*, 2017.

[28] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *IEEE S&P*, 2019.

[29] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, “Branch history injection: On the effectiveness of hardware mitigations against cross-privilege spectre-v2 attacks,” in *USENIX Security*, 2022.

[30] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer,” in *WOOT*, 2018.

[31] J. Wikner and K. Razavi, “Retbleed: Arbitrary speculative code execution with return instructions,” in *USENIX Security*, 2022.

[32] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, “PACMAN: Attacking ARM pointer authentication with speculative execution,” in *ISCA*, 2022.

[33] J. Wikner, D. Trujillo, and K. Razavi, “Phantom: Exploiting decoder-detectable mispredictions,” in *MICRO*, 2023.

[34] D. Trujillo, J. Wikner, and K. Razavi, “Inception: Exposing new attack surfaces with training in transient execution,” in *USENIX Security*, 2023.

[35] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *CCS*, 2018.

[36] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, “Speculative interference attacks: Breaking invisible speculation schemes,” in *ASPLOS*, 2021.

[37] O. Kirzner and A. Morrison, “An analysis of speculative type confusion vulnerabilities in the wild,” in *USENIX Security*, 2021.

[38] S. Wiebing, A. de Faveri Tron, H. Bos, and C. Giuffrida, “InSpectre gadget: Inspecting the residual attack surface of cross-privilege Spectre v2,” in *USENIX Security*, 2024.

[39] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, “Imp: Indirect memory prefetcher,” in *MICRO*, 2015.

[40] S. Ainsworth and T. M. Jones, “Graph prefetching using data structure knowledge,” in *ICS*, 2016.

[41] ——, “An event-triggered programmable prefetcher for irregular workloads,” in *ASPLOS*, 2018.

[42] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, “Augury: Using data memory-dependent prefetchers to leak data at rest,” in *IEEE S&P*, 2022.

[43] M. Cavus, R. Sendag, and J. J. Yi, “Informed prefetching for indirect memory accesses,” *ACM Trans. Archit. Code Optim.*, 2020.

[44] A. M. Kaushik, G. Pekhimenko, and H. Patel, “Gretch: A hardware prefetcher for graph analytics,” *ACM Trans. Archit. Code Optim.*, 2021.

[45] R. Cooksey, S. Jourdan, and D. Grunwald, “A stateless, content-directed data prefetching mechanism,” *ACM SIGPLAN Notices*, 2002.

[46] X. Yu, C. J. Hughes, and N. R. Satish, "Hardware prefetcher for indirect access patterns," Patent US20160188476A1, 2017.

[47] Intel, "Data dependent prefetcher," https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html, accessed on Jun 6 2024.

[48] Y. Tobah, A. Kwong, I. Kang, D. Genkin, and K. G. Shin, "SpecHammer: Combining Spectre and Rowhammer for new speculative attacks," in *IEEE S&P*, 2022.

[49] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "RAMBleed: Reading bits in memory without accessing them," in *IEEE S&P*, 2020.

[50] Intel, *Intel Architecture Instruction Set Extensions and Future Features*, March 2023.

[51] 504ENSICS Labs, "Lime," https://github.com/504ensicsLabs/LiME, accessed on Jun 6 2024.

[52] V. Lyubashevsky, L. Ducas, E. Kiltz, T. Lepoint, P. Schwabe, G. Seiler, D. Stehlé, and S. Bai, "Crystals-dilithium algorithm specifications and supporting documentation (version 3.1)," *NIST submission*, 2021.

[53] A. May and J. Nowakowski, "Too many hints - when LLL breaks LWE," in *ASIACRYPT*, 2023.

[54] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher," in *ASPLOS*, 2023.

[55] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *USENIX Security*, 2019.

[56] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Binoculars: Contention-based side-channel attacks exploiting the page walker," in *USENIX Security*, 2022.

Appendix A. Flushing the page walker caches

Recall from Section 3.2 that to construct page walker cache order oracle, the attacker requires the ability to selectively flush partial translations of a secret from the page walker caches. To achieve this, our attacks rely on filling up the target page walker cache level with unrelated partial translations, effectively evicting all existing cache entries including the secret's partial translation. Specifically, to evict page walker cache level i, we access n addresses with different PL_i bits, progressively increasing n until a new cache access is observed during the attack (cf. Figure 6). To flush the 1st level page walker cache (also known as the TLB), we access n addresses spaced 4 KB apart. For the 2nd, 3rd, and 4th level page walker caches, we access n addresses spaced 2 MB, 1 GB, and 512 GB apart, respectively. Our attacks determine the value of n at runtime and do not rely on knowing the exact page walker cache sizes.

Appendix B. Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper combines speculative execution primitives with data prefetchers in CPUs to leak arbitrary secrets. Specifically, it demonstrates that modern CPUs could be forced to dereference arbitrary secrets as pointers though mis-speculation and the side effects of the page walk process during the referencing can be used to infer all the bits of the secret except the 6 LSB bits corresponding to the cacheline offset. The paper demonstates the feasibility of this attack on Intel CPUs.

B.2. Scientific Contributions

- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Independent Confirmation of Important Results with Limited Prior Research

B.3. Reasons for Acceptance

1) This paper identifies an impactful vulnerability: This paper for the first time demonstrates the ability to leak arbitrary secrets by exploiting a combination of speculative execution and data prefetching behavior in modern CPUs. The authors also describe a novel differential prime and probe attack that relies on inferring cache side effects by capturing snapshots before and after the sensitive code has executed. The exploration of the page walk caches and the demonstration of how one can systematically infer secret bits by terminating the page walk at different levels is also noteworthy.
2) This paper provides a valuable step forward in an established field and this paper independently confirms results with limited prior research: This paper for the first time demonstrates leaking of secrets that looks like invalid addresses (e.g., do not comply with canonicality checks). This is a significant improvement over the closest related work on side channels based on data dependent prefetchers (GoFetch, Augury) as well as page walk information (AnC).