Leaky Address Masking: Exploiting Unmasked Spectre Gadgets with Noncanonical Address Translation

Mathé Hertogh*, Sander Wiebing* and Cristiano Giuffrida*
*Vrije Universiteit Amsterdam, The Netherlands
{m.c.hertogh,s.j.wiebing,c.giuffrida}@vu.nl

***Abstract*—Linear Address Masking (LAM) is a recently announced Intel feature that enables the CPU to mask off some upper bits before dereferencing a 64-bit pointer. The key idea behind LAM (as well as the similar Upper Address Ignore or UAI from AMD), is to allow software to efficiently make use of untranslated bits of 64-bit linear addresses for metadata. The assumption is that, with LAM (or UAI) features enabled, one can implement fast security (e.g., memory safety) checks and ultimately improve security of production systems.**

In this paper, we challenge this assumption and show that LAM features can actually *degrade* security in production by dramatically increasing the Spectre attack surface. To support this claim, we present a new Spectre covert channel based on *noncanonical address translation* and address key challenges to implement it in practice. For instance, we exploit properties of modern TLBs to craft a reliable signal and LAM features to (crucially) bypass canonicality checks. Moreover, we show that, unlike classic Spectre covert channels, ours unlocks generic (or *unmasked*) Spectre gadgets encoding high-entropy secrets as dereferenced pointers. Unlike classic (or *masked*) gadgets, we show the latter escape deployed mitigations and are pervasive in high-value targets such as the Linux kernel. To showcase the new attack surface, we present an end-to-end exploit for Spectre based on LAM (SLAM) targeting upcoming Intel CPUs. We specifically focus on the BHI Spectre variant and show that, despite mitigations believed to eradicate the attack surface, our exploit can abuse a variety of gadgets in the latest Linux kernel and leak the root password hash within minutes from kernel memory. We conclude by evaluating mitigations.

1. Introduction

Since the original Spectre [1] and Meltdown [2] disclosure in 2018, transient execution vulnerabilities have been increasingly gaining momentum. Five years, several disclosed variants, and even more deployed mitigations later, a key question still troubles researchers and practitioners: "*What is the residual attack surface for last-generation systems?*". For Meltdown-like vulnerabilities (e.g., L1TF [3], MDS [4], [5], [6], [7], [8], etc.) fully mitigated in hardware, the answer is relatively well-understood (i.e., "*none*", modulo the occasional mitigation flaws [8]). For Spectre, on the other hand, the answer is far from trivial.

```
void masked_gadget(long *secret) {
  array[(*secret & 0xff) * 4096];
}
void unmasked_gadget(long **secret) {
  **secret;
}
```

Figure 1: Masked and unmasked Spectre disclosure gadgets. The attacker controls `secret` during speculative execution.

Indeed, since Spectre vulnerabilities remain not fully mitigated in hardware, the residual attack surface depends on unintentionally exploitable code snippets (or *disclosure gadgets*) in the victim software. Typically, attackers only need to find one gadget to disclose *secret data*. They can then rely on direct/indirect branch misprediction to lure the victim into speculatively executing the gadget, the latter *accessing* and then *transmitting* the secret over a microarchitectural covert channel. Figure 1 (top) exemplifies a classic ("*masked*" [9]) gadget, with `*secret` (e.g., out-of-bounds) data *masked* down to 8 bits, encoded as an index of a small linear `array`, transmitted when accessing the corresponding array element's CPU cache line, and ultimately received by the attacker via a cache attack such as Flush+Reload [10].

With deployed mitigations reducing exploitable branch targets (e.g., by fencing direct branches [1] or tagging/flushing indirect branches [11]) as well as guarding sensitive array accesses to hinder masked gadgets (e.g., *array_index_nospec* [12]), finding classic Spectre gadgets or variations [11], [13], [14], [15] one can exploit in practice is a tall order. This is the case even for the large codebases of high-value victims such as the Linux kernel. Indeed, state-of-the-art kernel gadget scanners generally only report *potentially* exploitable gadgets [9], [11], [15], [16] or find exploitable ones that depend on other mitigated Meltdown-like vulnerabilities such as MDS [16], [17]. Existing end-to-end exploits, on the other hand, need to resort to software vulnerabilities [13], language features such as (unprivileged) eBPF [11], [15], or vulnerable older-generation microarchitectures [14]. As such, common wisdom suggests that the residual Spectre attack surface is thin in practice.

In this paper, we challenge common wisdom and uncover a significant new attack surface for user-to-kernel

Spectre attacks on upcoming Intel/AMD CPUs[1]. Specifically, we move away from classic Spectre gadgets and study "*unmasked*" gadgets [9], demonstrating their practical exploitation for the first time. Figure 1 (bottom) exemplifies an unmasked gadget, with 64-bit `*secret` data encoded as a dereferenced pointer. As we will show, unmasked gadgets originate from widespread *pointer-chasing* code patterns, and, as such, are abundant in modern kernels such as Linux.

Nonetheless, exploiting unmasked gadgets is challenging, to the point that their practical relevance has been often dismissed in the past [9]. Indeed, since these gadgets interpret an arbitrary high-entropy secret as a pointer and use a simple dereference for transmission, practical exploitation is out of reach for classic cache covert channels. First, the secret pointer may happen to encode an invalid address, such as a noncanonical or unmapped address, whose dereference may be unable to fill a valid cache line and transmit the secret. Second, even if the secret happens to encode a valid address, the pointer dereference may fill a cache line anywhere within (huge and noncontiguous) valid 64-bit virtual address space, far exceeding the small linear array (and the CPU cache size) required by classic cache covert channels.

To address these challenges and show unmasked gadgets are exploitable in practice with few constraints, we devise a new Spectre covert channel based on a number of key insights. First, we move away from classic cache covert channels and opt for an *address translation* one. To this end, we consider different translation vectors and show that, due to their properties, modern TLBs are the ideal choice, simultaneously yielding the best efficiency and the largest attack surface. Unlike prior TLB covert channels [6], [18], [19], ours abandons the classic linear array design and adopts an efficient Evict+Reload construction [20] for the transmission, matching the reliability of classic cache covert channels. Second, we extend our covert channel to support *noncanonical address translation*. As we will show, this is possible by abusing *Linear Address Masking* (LAM, or UAI in AMD parlance) features [21] in upcoming Intel/AMD CPUs, which mask off crucial upper pointer bits normally subject to canonicality checks. We further show our covert channel can also target older-generation AMD CPUs that feature lazy canonicality checks. Finally, we devise a combination of *sliding* and *just-in-time remapping* techniques to reduce the entropy of our covert channel, enabling byte-by-byte disclosure and ultimately practical exploitation.

To evaluate the new attack surface, we use our covert channel to mount a practical user-to-kernel exploit for Spectre based on LAM (SLAM) against Linux. For our analysis, we specifically focus on the BHI Spectre variant [11], given that (i) assessing the residual BHI attack surface post unprivileged eBPF is an open and pressing question, having recently persuaded Intel to conduct the first large-scale, in-depth gadget analysis campaign from a vendor [9]; (ii) despite a dedicated gadget scanner and the extensive manual effort, said analysis found no attack surface of concern—after focusing on classic masked Spectre gadgets. In contrast, as we will show, even the simple gadget scanner we develop reveals hundreds of *practically* exploitable gadgets (out of 16,046 potential ones) to implement BHI exploits based on our TLB covert channel. As a concrete demonstration, we exploit 7 such gadgets in end-to-end SLAM exploit instances able to leak arbitrary ASCII characters from Linux kernel memory on upcoming Intel/AMD CPUs. Our exploits can leak the root password hash on the latest Ubuntu within minutes. We conclude by evaluating mitigations.

In summary, our contributions are:

- The first security analysis of Intel LAM / AMD UAI, with concrete evidence upcoming LAM features can degrade (rather than improve) security.
- An analysis of translation-based covert channels, resulting in the first TLB Evict+Reload primitive able to disclose information from a privileged victim.
- The first (*noncanonical address translation*) covert channel enabling practical exploitation of unmasked Spectre gadgets.
- The first in-depth unmasked Spectre (BHI) gadget analysis for Linux and the resulting practical end-to-end SLAM exploit to leak the root password hash.
- Investigation and evaluation of mitigation options.

Availability. Code and additional information about SLAM is available at https://vusec.net/projects/slam.

2. Background

2.1. Address Translation

Memory management in modern CPUs uses a *paging* organization. Loads and stores operate on *virtual addresses*, which are *translated* by the Memory Management Unit (MMU) into *physical addresses* (normally referencing DRAM). Software has control over such virtual-to-physical *address translation* via multi-level *page tables*, i.e., a memory-resident radix tree mapping virtual to physical addresses, at the *page* (e.g., 4 KB) granularity. Upon every (virtual) memory reference, the MMU performs a *page table walk*, i.e., a radix-tree lookup to find the corresponding physical address, after which the actual data can be fetched. Walking the page tables requires extra memory accesses, since a *page table entry* (PTE) must be fetched at each page table level. PTE fetches use the same memory subsystem and hence the same CPU caches as normal data accesses. As an additional optimization, both full and partial address translations are cached in dedicated MMU caches, known as *Translation Lookaside Buffers* (TLBs) and *Translation Caches* (respectively). Page tables also support permission bits in each PTE, with software able to mark pages as user/supervisor, read/write/execute, etc. Modern operating systems deploy SMAP [22], a hardware feature preventing the kernel from accessing user memory.

2.2. Linear Address Masking

Modern x86_64 platforms support 48-bit or 57-bit virtual addresses with 4-level or 5-level page tables (respec-

[1]See Section 14 for impact also on future ARM CPUs.

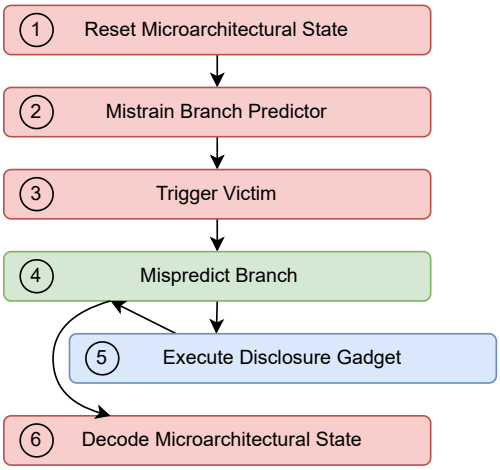

Figure 2: Overview of Spectre attacks, with architectural execution of the attacker (red) and architectural / transient execution of the victim (green / blue).

tively). Since pointers encode 64-bit virtual addresses, the upper (16 or 7, respectively) bits are irrelevant for address translation and instead required to be copies of the top translated bit (47 or 56, respectively)—conventionally set for kernel addresses. Addresses complying to this requirement are said to be in "*canonical form*". Accessing a noncanonical address normally results in an exception, an inconvenience for memory sanitizers [23] and mitigations [24], [25], [26] which tag unused upper pointer bits to store metadata.

To address this problem, upcoming Intel/AMD CPUs implement support to mask some upper pointer bits before translation, loosening classic canonicality checks to accommodate software-managed tagged pointers. Such features are branded as Linear Address Masking (LAM) on Intel [21] and Upper Address Ignore (UAI) on AMD [27]. We elaborate on LAM/UAI details as well as their ability to unlock SLAM exploitation in Section 6.

2.3. Spectre Attacks

Spectre attacks [1] abuse modern processors' inability to roll back microarchitectural state modified by speculative execution. This allows an attacker to lure the CPU into *speculatively* executing code that should never run *architecturally*, inducing speculative accesses to secret data, and encoding the secret into microarchitectural state which the attacker can later decode (the *covert channel*). The most common covert channel is Flush+Reload [10], which exploits CPU cache lines in a *reload buffer* shared between the attacker and the victim.

Figure 2 shows an overview of Spectre attacks. Attackers ① reset the microarchitecture to some known state (e.g., flush their reload buffer from the cache), ② prepare their speculative control-flow hijack by mistraining a branch predictor, and ③ trigger victim execution with some malicious input (e.g., syscall). During the execution of the victim, the CPU mispredicts the mistrained branch ④ and speculatively executes a *Spectre disclosure gadget*, using attacker-controlled data. Traditionally, attackers target a "*masked*" gadget [9], encoding the secret as an index into an accessed array (Figure 1)—i.e., the reload buffer. In general, the (disclosure) gadget ⑤ encodes the secret into some microarchitectural state, which attackers later ⑥ decode to leak the secret (e.g., by timing accesses to the different cache lines of the reload buffer to find the accessed array index).

Spectre attacks can hijack different types of branches. For example, Spectre-v1 hijacks conditional direct branches, while Spectre-v2 hijacks indirect branches. A recent type of Spectre-v2 attack is Branch History Injection (BHI) [11]. BHI speculatively hijacks an indirect *victim branch*, by poisoning the Branch History Buffer (BHB) with a colliding branch history. This lures the branch predictor into incorrectly predicting the *victim branch* and transferring control flow to the destination of another (colliding) *target branch*.

3. Threat Model

We consider a typical Spectre local exploitation scenario, with an attacker controlling an unprivileged user process on a victim machine and targeting a user-to-kernel Spectre attack to leak secrets from kernel memory. We specifically target the (secret) root password hash, as done in prior work [4], [11], [13], [14]. We assume a victim machine equipped with LAM features and running the latest Linux kernel with all the defenses against transient execution vulnerabilities enabled. We also assume other (e.g., memory safety) vulnerabilities are mitigated by orthogonal defenses.

4. SLAM

At a high-level, SLAM exploits follow the same workflow of classic Spectre exploits (Figure 2). The key difference is that SLAM exploits *unmasked* Spectre gadgets, which encode the secret as a dereferenced pointer (i.e., *secret pointer*). As detailed later, such gadgets are abundant as they often originate from common *pointer-chasing* code patterns. A typical real-world example is the Linux kernel gadget listed in Figure 3. As shown in the figure, similar to classic masked gadgets, a speculative load (via the attacker-controlled `iocb` argument) reads secret data on Line 4. However, unlike classic masked gadgets, the secret is then interpreted as a pointer (`f`) and directly dereferenced on Line 5, a poor match for classic cache *covert channels*.

To understand the complications, let us consider a baseline exploitation scenario. Suppose a user-to-kernel attacker wants to craft a 1-bit information disclosure primitive to leak whether the secret data matches a predetermined, valid kernel pointer (Figure 4, top). For this scenario, a classic cache covert channel is sufficient. Indeed, the attacker can ensure the cache line backing the target kernel pointer is nonpresent (e.g., by walking a corresponding eviction set [28]), trigger speculative execution of the gadget, and then probe the cache to check if the target cache line is

```
1  ssize_t kernfs_fop_read_iter(
2                  struct kiocb *iocb,
3                  struct iov_iter *iter) {
4    struct file *f = iocb->ki_filp;
5    struct seq_file *sf = f->private_data;
```

Figure 3: A typical SLAM disclosure gadget in the Linux kernel, with a call to `kernfs_of` inlined for readability. An attacker speculatively controlling `iocb` can lure the kernel into reading (Line 4) and disclosing (Line 5) data.

now present (e.g., a la Prime+Probe [29]). However, while this simple *baseline primitive* may be sufficient to break KASLR (i.e., with the attacker repeatedly trying to guess a target randomized kernel pointer until they see a signal in the cache), generalizing this primitive to generic secrets such as the root password hash is a tall order.

Indeed, without strong assumptions on the secret data, the secret pointer may not reference valid kernel memory. Figure 4 (bottom) shows a running example encoding for the first few bytes ("root:y") of the secret root password hash in Linux' `/etc/shadow` file. As the figure shows, the secret is encoded with a *user*, *noncanonical* address. In fact, we note that ASCII strings (i.e., our target) are *always* encoded as *user* pointers, as their bytes always have the uppermost bit unset, including the top translated bit (yellow in figure). Moreover, out of the whole 64-bit virtual address space, less than 0.01% is canonical. Since both user and noncanonical pointer dereferences raise exceptions (i.e., due to SMAP and canonicality checks, respectively) and do not normally fill cache lines even on speculative paths, the secret leakage surface of our unmasked gadget is thin for classic cache covert channels.

Furthermore, even if the secret happened to encode a valid pointer, the high (no less than 47-bit) entropy of the secret pointer would pose an additional problem. While masked gadgets typically leak 8 bits of secret data at a time, conveniently encoded in an index of a 256-element reload buffer, unmasked gadgets cannot easily fit the small-linear-array model. Indeed, the distribution of valid secret pointer values is large and highly nonlinear, also well exceeding the occupancy of modern CPU caches.

Hence, starting from a baseline 1-bit valid kernel pointer disclosure primitive, SLAM tackles three main challenges:

- (C1): How do we extend our disclosure primitive to user pointers? As we shall see, SLAM addresses this challenge with *address translation* covert channels.
- (C2): How do we extend our disclosure primitive to noncanonical pointers? As we shall see, SLAM addresses this challenge by bypassing canonicality checks with *LAM features*.
- (C3): How do we generalize our disclosure primitive to actually leak secrets? As we shall see, SLAM addresses this challenge by a combination of *sliding* and *just-in-time remapping* techniques.

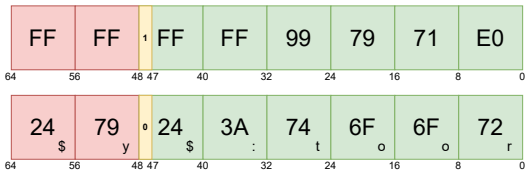

Figure 4: A valid kernel pointer (top) vs. root password hash bytes encoded as a pointer (bottom) on 4-level paging systems. Bits 63:48 (red) are subject to canonicality checks. Bit 47 (yellow) is set (unset) for kernel (user) pointers.

5. Leaked in Translation

To address (C1), we need to expand the leakage surface of our baseline 1-bit information disclosure primitive (i.e., leaking whether the secret is a predetermined, valid kernel pointer) to user pointers. To this end, we need a covert channel capable of encoding the secret into microarchitectural state upon a user pointer dereference. This is infeasible for classic cache covert channels, since user pointer dereferences are invalid (and thus unable to complete and fill cache lines) in kernel mode due to Supervisor Mode Access Prevention (SMAP) [22]. This is the case architecturally [22] and (although speculative SMAP behavior remains "officially" undocumented [30]) also microarchitecturally [31].

To address this challenge, the key insight is that we need a covert channel based on microarchitectural state that gets involved *before* SMAP checks kick in. And since, for SMAP checks to kick in, the MMU needs to first determine whether the kernel is dereferencing user memory, we turn our attention to the *address translation* process. Indeed, the MMU needs to complete address translation in order to locate the appropriate translation entry (PTE) and check the corresponding *supervisor* bit (unset for user memory).

To confirm this behavior, we set up an experiment with the unmasked Spectre gadget in Figure 3. We speculatively executed the gadget in the kernel while instructing it to read and dereference a secret encoded as a (valid and present) user address. As expected, on all of our tested microarchitectures (detailed later), we observed the backing user cache line not to be filled by the gadget, confirming the infeasibility of the cache covert channel. At the same time, using existing microarchitectural attacks [32] we clearly observed a signal for translation-related activity, confirming address translation completes before SMAP checks can get a chance to dismiss the user pointer dereference as invalid.

Building on these results, our next step is to craft an *address translation covert channel* for secrets encoded as user pointers. We observe that, other than being able to bypass SMAP checks, such covert channel has a number of other advantages compared to classic cache covert channels. For example, address translation is agnostic to the particular memory access type. This can increase the number of available gadgets compared to classic cache covert channels, which typically rely on regular load instructions and may be impaired by store instructions [18]. Finally, since address translation leaves persistent traces in the many microar-

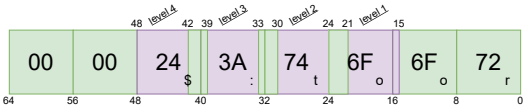

Figure 5: At each page table level, 6 bits (purple) of the virtual address can be retrieved via the corresponding PTE.

chitectural components involved in the translation process, there are multiple options to craft our covert channel. We consider the main options in the next subsections. For simplicity, we develop our analysis along our running example on 4-level paging systems (Figure 4), but our results directly extend to 5-level paging systems.

5.1. PTE Probing

During address translation, the MMU walks page tables and fetches PTEs from memory via the same data cache hierarchy as regular memory accesses [32]. Bits 47:39 determine the PTE of the secret pointer at the (4th level) root page table. As 8 PTEs, of 8 bytes each, fit into one 64-byte cache line, bits 47:42 determine the PTE's cache line. Hence, by inferring which of the 64 cache lines constituting the root page table gets fetched, via a set-granular cache attack such as Prime+Probe or Evict+Time [32], an attacker can retrieve bits 47:42 of the secret pointer. This allows the attacker to tell whether a given secret pointer was dereferenced (baseline 1-bit disclosure primitive) or even discriminate between different secrets. This idea generalizes to the lower-level page tables, provided that the PTEs of previous levels are valid (i.e., present and with the correct permissions). Figure 5 shows which bits can be retrieved at each level for our (canonicalized) running example.

An advantage of the PTE probing covert channel is that one can retrieve (some) secret bits even for nonpresent addresses as long as partial page table (e.g., top-level) information is present. At the same time, a disadvantage is that page table walks require a relatively large speculation window, as multiple cache misses need to fit in the window. In addition, the reliance on cache set probing results in a noisy covert channel, especially during kernel execution [13]. Finally, for deep page table walks, it is challenging to match up signals in multiple cache sets with the different PTE levels [32]. To address these shortcomings, we turn to the TLB next.

5.2. TLB Probing

To build a covert channel based on the TLB, we need to first confirm the TLB incurs microarchitectural state changes upon a speculative load incurring an SMAP exception. On x86, TLBs are known not to perform *negative caching*, i.e., invalid entries produced by a page table walk are not cached in the TLB. This behavior is documented for nonpresent entries [33], but not for other erroneous entries. To investigate the behavior, we repeated the same experiment as above (i.e., letting the kernel speculatively dereference a user address) multiple times for both present/nonpresent user addresses after first flushing the backing TLB entry. With the help of existing microarchitectural attacks [34], we confirmed the behavior for nonpresent addresses: a load referencing a nonpresent user address did not fill the TLB (i.e., no TLB hit in repeated experiments). However, for the present user address case we observed the opposite behavior, with the speculative load filling the TLB despite the SMAP fault (i.e., TLB miss on the first repetition, TLB hit on the second one).

Armed with this knowledge, one can build a TLB probing covert channel for secret user pointers. Specifically, one can rely on a set-granular TLB attack such as TLB Prime+Probe or Evict+Time [34] to retrieve the TLB set accessed by the Spectre gadget and the corresponding bits of the secret pointer. In contrast to PTE probing, this covert channel requires shorter windows—since we can control and reduce the work done on the TLB miss encountered by the gadget—and eliminates the need to probe multiple sets at the time. At the same time, this covert the channel can only operate on present user addresses and is even more noisy—since TLB set page-granular collisions are much more common than cache set collisions. We later detail how to handle the former complication. We handle the latter next.

5.3. TLB Reloading

To eliminate the need for TLB set probing, the key insight is that, since we explicitly target *user* addresses accessed by the kernel (due to our ASCII string target), the accessed TLB entry is *shared* between the attacker (user) and the victim (kernel). As such, we can adapt the Evict+Reload cache attack [20] to the TLB, directly retrieving the TLB entry accessed by the Spectre gadget and the corresponding bits of the secret pointer. This is done by: (i) walking TLB eviction sets to evict the TLB entry backing our target user address in userland [34], [35], (ii) triggering the speculative execution of the gadget in the kernel, and (iii) reloading the target TLB entry. To reload the TLB entry, the attacker needs to measure the latency of the target user address translation.

To this end, one option is to rely on the translation-dependent timing of the *prefetch* instruction(s) [36], [37], although we observed unreliable timings on our AMD testbed. A more general solution is to measure the time to complete a load referencing the target user address. However, since such time is both translation- and data fetch-dependent, care should be taken to properly isolate the translation latency. For this purpose, we need to control the source of the data fetch and select the one that maximizes the signal.

Selecting the cache level. In our experiments, we determined L2 data fetches to be consistently optimal. Indeed, as shown in Figure 6, the virtually indexed L1 cache is looked up in parallel to TLB, thereby masking the TLB signal. Higher data cache levels are physically indexed, serializing TLB and cache access latencies and yielding a better signal. However, the higher the selected cache level (or in the worst case, DRAM), the higher the jitter caused

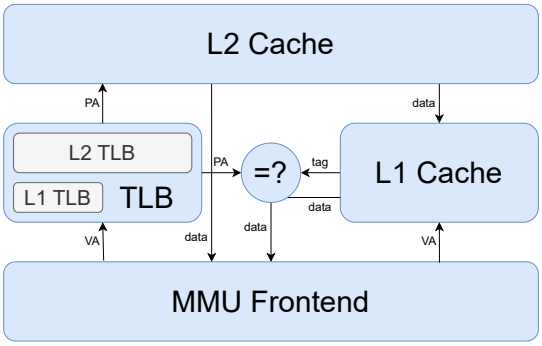

Figure 6: High-level overview of TLB and data caches.

Figure 7: Depending on the page size, a number of virtual address bits (purple) can be retrieved via the TLB.

by the more complex microarchitectural geometry (e.g., last-level cache slices). As such, we found L2 to be a sweet spot.

Selecting the TLB level. As depicted in Figure 6, modern MMUs commonly deploy two-level TLBs [35], hence attackers can get their signal via either L2 or L1 TLB misses. In the former scenario, one can evict both (L1 and L2) target TLB entries and calibrate the reload step's timings to distinguish between a L1 TLB hit and a L2 TLB miss. In the latter scenario, one can evict only the target L1 LTB entry while preserving L2 and instead focus on a L1 TLB hit versus L1 TLB miss (L2 hit) signal. Indeed, after missing L1 and hitting L2, the Spectre gadget causes the MMU to fill the L1 TLB entry, yielding the signal. For our purposes, we found the latter scenario to be preferable for a variety of reasons. First, the lack of L2 TLB misses eliminates page table walk jitter (e.g., cached vs. uncached PTEs), improving the signal. Second, as detailed in Section 11.2, the small L2 TLB hit latency results in short speculation windows, which can help bypass certain mitigations [38]. Finally, as detailed in Section 6.2, (L2) TLB hit-based gadgets extend the attack surface to additional (existing) AMD microarchitectures.

Selecting the page size. Modern operating systems support 4 KB, 2 MB, and 1 GB pages. Modern TLBs lay out their entries in separate partitions accordingly [39]. As such, by selecting a different page size for our target user address to reload, we can use a different part of the TLB as our covert channel. Figure 7 shows which bits can be retrieved for our (canonicalized) running example for the different page sizes (although 1 GB pages are often restricted in practice). The smaller the page size, the larger the number of secret pointer bits one can retrieve, although this is not crucial as we shall see in Section 7. A more pressing concern is noise isolation. One should select a page size that is as infrequently used as possible by the kernel, in order to minimize interference with the TLB partition used by the covert channel and thus maximize the signal. Since the kernel uses 2 MB pages for its own text and data sections as well as (mostly) 1 GB pages for its direct map of physical memory, we select 4 KB pages for our covert channel.

5.4. Summary

We considered a number of translation covert channels for SLAM. We focused on the main options, but variations are possible. For instance, one could rely on other microarchitectural components, such as *page table caches* [40] to reduce the noise on the PTE probing signal, or *translation caches* [40] to extend our TLB covert channel to nonpresent addresses. In practice, the microarchitectural details of modern TLBs are well understood and we did not find any serious limitations that deterred their use for SLAM.

C1 Solution. SLAM uses a *TLB Evict+Reload* covert channel, with 4 KB TLB entries and a L1 vs. L2 hit signal. This provides a low-noise, low-latency covert channel to extend the leakage surface of our baseline 1-bit pointer disclosure primitive to *present user addresses*.

6. Canonicalizing Secrets

To address **C2**, we need to expand the leakage surface of our existing 1-bit user pointer disclosure primitive to noncanonical user pointers. This is important, as so far we have only dealt with canonical user pointers, while secret ASCII strings—barring those with `NUL` characters in strategic positions—*always* encode to noncanonical ones. The latter cannot be normally processed by our TLB covert channel, as translation hinges on successful canonicality checking. To this end, we consider different mechanisms to "*canonicalize*" secrets on Intel and AMD platforms.

6.1. Intel Platforms

On upcoming Intel platforms (e.g., Sierra Forest, Grand Ridge, Arrow Lake, and Lunar Lake [21]), we turn to LAM for our purposes. With LAM enabled, some upper pointer bits are "masked" upon pointer dereference. LAM has two modes: (i) LAM48, masking 15 upper bits (62:48) for 4-level paging systems; (ii) LAM57, masking 6 upper bits (62:57) for 5-level paging systems. In both cases, the upper

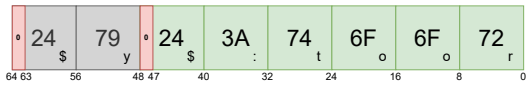

Figure 8: Intel LAM masks away bits 62:48 (grey), reducing the canonicality check to the equality of bits 47 and 63 (red).

nontranslated bits except the most-significant bit are masked, i.e., copied from the top translated bit ([62:48] := [47], for 4-level paging), *before* address translation occurs [21]. Since the masked bits are made canonical by LAM, the original upper pointer bits are no longer subject to canonicality checks and can thus store arbitrary values. Bit 63 is the only special case, required to match the top translated bit to avoid a noncanonical address exception.

Bit 63 is also used as a "supervisor" bit for LAM to distinguish kernel from user pointers. Indeed, LAM can be enabled separately for user and/or supervisor pointers via control registers. Software support merged in recent Linux kernel versions [41] enables LAM only for user pointers. This allows user processes to enable LAM for their own (tagged) pointers. With LAM enabled, user pointers dereferenced by *kernel* code are still masked, as LAM honors the zero bit 63 but ignores the privilege level.

The latter property allows a LAM-enabled user process to pass noncanonical user pointers to the kernel and the kernel to later dereference them as part of in-kernel syscall handling. This is important to simplify kernel support for LAM, as user pointers can be dereferenced "as-is" by the kernel, eliminating the need for the kernel to explicitly mask them. However, this property is also crucial to unlock SLAM exploitation. Indeed, as shown in Figure 8, LAM effectively disables the canonicality check for all the upper bits but bit 63 of the secret pointer dereferenced by the kernel. In other words, under LAM, the canonicality requirement is reduced to bit 63 and the top translated bit of a pointer being equal. However, since SLAM targets secret ASCII strings, this invariant always holds for our secret user pointers. We conclude that our TLB covert channel can bypass canonicality checks with Intel LAM.

6.2. AMD Platforms

UAI is AMD's LAM variant on upcoming AMD platforms. UAI is simpler than LAM and somewhat closer to ARM's Top Byte Ignore [42], with the MMU simply ignoring the most-significant 7 bits of a virtual address during translation. Since only 7 bits are ignored, on 4-level paging systems 9 bits (56:48) are still affected by canonicality checks. This effectively hinders SLAM exploitation—at least for generic ASCII strings. However, on 5-level paging systems those 9 bits are involved in address translation (indexing the level-5 page table), allowing our TLB covert channel to bypass canonicality checks with AMD UAI.

We also considered SLAM exploitation on existing AMD microarchitectures vulnerable to *Transient Execution of Noncanonical Accesses* (TENA) [43], [44]. Indeed, as shown in Figure 9, such 4-level paging microarchitectures

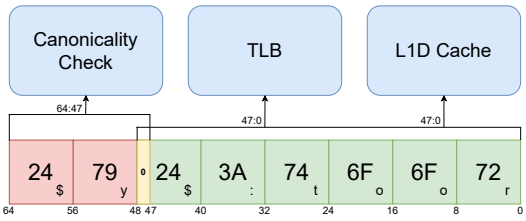

Figure 9: The canonicality check races against the TLB and the data cache: bits 47:0 (green and yellow) get passed to the TLB and the data cache, and in parallel bits 63:47 (yellow and red) are passed to the canonicality checker.

parallelize translation (+ data fetch) and canonicality checks. Specifically, the CPU uses the lower 48 bits of the virtual address to consult the TLB and L1 data cache, as well as the upper 17 bits to check canonicality in parallel. As a result, the TLB and data cache *ignore* the upper 16 bits of the virtual address, hence a noncanonical address can initiate a memory access to its canonical counterpart (same lower 48 bits). Moreover, this creates a microarchitectural race between the canonicality check and the memory access: the target data may be transiently forwarded to later instructions in the pipeline, despite the noncanonical target address.

Prior work has exploited the latter property to craft Meltdown-type gadgets that speculatively load data from a noncanonical address and then leak said data via a classic cache covert channel [43]. According to the authors, their analysis did not reveal exploitable instances of such gadgets in practice. In contrast, with SLAM, we want to show that transient noncanonical accesses *do* have practical impact when used to support the *address translation covert channel* of an unmasked Spectre gadget. For this to happen, we need to ensure the requirements are satisfied. According to the AMD documentation, transient noncanonical accesses are possible only when address translation incurs a TLB hit [43], but no requirements on the particular TLB level are specified. Our proposed TLB covert channel does incur a TLB hit, but only in L2 (and not in L1) TLB by construction.

To uncover whether L2 TLB hits are sufficient, we set up an experiment with the unmasked Spectre gadget in Figure 3. We speculatively executed the gadget in the kernel, while instructing it to read and dereference a secret encoded as a noncanonical (present) user address and performing L1 TLB Evict+Reload for the backing TLB entry. From the affected AMD platforms [43], we tested on Ryzen 7 2700X, available in our lab. Our results revealed a signal for the target address, confirming the gadget performs the transient noncanonical access upon L2 TLB hit and reinserts the backing entry into the L1 TLB. We conclude that our TLB covert channel can successfully support noncanonical secret user pointer dereferences on microarchitectures affected by TENA, even in absence of hardware masking features.

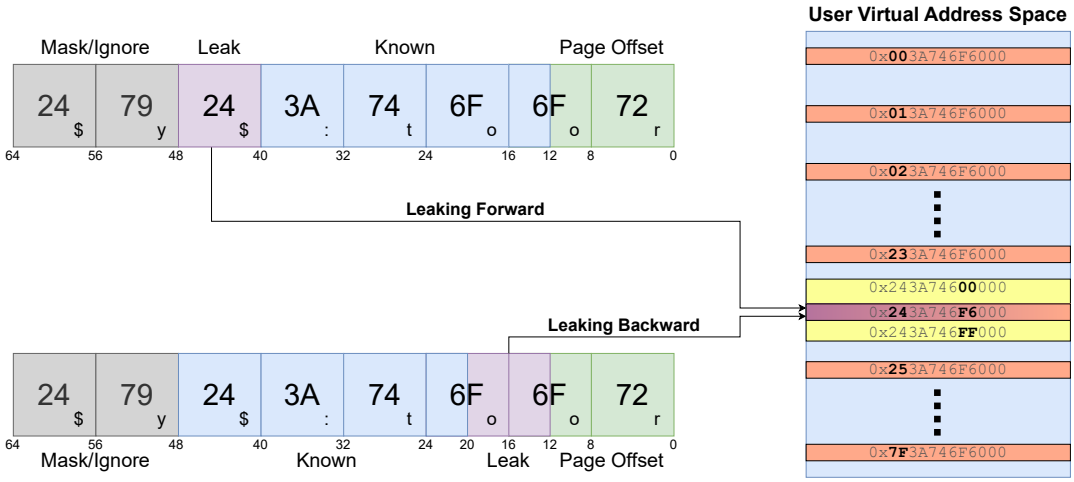

Figure 10: Left: secret leakage using forward (top) and backward (bottom) sliding. Vulnerable hardware canonicalizes the grey bits, and translation ignores the green bits. With the blue bits known, SLAM leaks the remaining purple bits. Right: just-in-time reload buffer for forward leakage (noncontiguous, in orange) and backward leakage (contiguous, in yellow).

6.3. Summary

We considered different mechanisms to canonicalize 64-bit secret pointers for our TLB Evict+Reload covert channel. On both Intel and AMD platforms, we successfully found canonicalization strategies for our purposes.

C2 Solution. SLAM can rely on LAM / UAI features to canonicalize noncanonical user pointers. Similar primitives exist on AMD systems vulnerable to TENA. These primitives expand the leakage surface of our 1-bit pointer disclosure primitive to *noncanonical user addresses*.

7. Leaking Secrets

To address C3, we need to generalize our 1-bit noncanonical user pointer disclosure primitive to one able to actually leak secrets. In other words, we need to turn our 1-bit TLB covert channel—able to test for the presence of a predetermined secret—into a generic N-bit covert channel—able to disclose arbitrary (ASCII) 64-bit secrets. A simple 1-bit covert channel is clearly impractical for the task (as we would need to test all the possible secret values to leak), but so is a high-entropy one (as we may lack microarchitectural state to encode the possible values all at once).

To address this challenge, we first observe that the techniques introduced in the previous sections already provide significant entropy reduction for our 64-bit secret pointers. Specifically, secret canonicalization eliminates entropy in the upper bits, and our TLB 4 KB page-granular covert channel eliminates entropy in the lower 12 bits. As also shown in Figure 7, this still leaves us with 36 bits of entropy (on 4-level paging systems). However, since we leak ASCII data—every byte's top bit is zero—the entropy is further reduced to 31 bits. Such residual entropy leads to a reload buffer of 8 TB, far exceeding the size of modern TLBs, as well as being practically unmanageable for an attacker. We tackle these two issues in order next.

7.1. Reducing Entropy with Sliding

To ensure the reloaded pages fit the size of modern TLBs, we want to reduce the entropy of the secret to that of a single byte per gadget iteration. To this end, we *slide* the pointer referencing secret data byte-by-byte across iterations, causing the secret retrieved by the current iteration to contain only *known* bytes from the previous iterations except one (new and *unknown*) byte. Figure 10 (top-left) exemplifies an iteration of such byte-by-byte leakage strategy on 4-level paging systems, with bits 39:12 (blue) known and bits 47:40 (purple) to be leaked. Since bit 47 is always zero due to ASCII, the entropy per iteration is reduced to 7 bits. Hence, we need a reload buffer of only 128 pages. Similar to prior sliding techniques [4], [13], [14], [45], we also need a way to kick start the strategy for the first iteration with some data known a priori. While one can in principle exploit memory massaging to colocate arbitrary secrets with known data [45], ASCII strings typically already contain known data in-band. This is indeed the case for our target `/etc/shadow` file, which consistently interleaves known (username) with secret (password) data.

7.2. Just-in-time Reload Buffer Remapping

Although we have reduced the number of reloaded pages per iteration, the total possible range is still 8 TB scattered across virtual address space. With demand paging, one could map a single "*sparse*" reload buffer spanning the entire user address space for this purpose. However, this simple strategy incurs many (i.e., up to 128) costly page faults per iteration and unbounded memory consumption (or costly cleanups).

To address these issues, we instead `mmap` only the range sufficient to cover all 128 pages of the first-iteration and each time `mremap` the reload buffer *just-in-time* to cover the 128 pages needed by each subsequent iteration. Initially, this results in a *noncontiguous* reload buffer, as illustrated in orange in Figure 10 (right) for our example pointer: the attacker already knows "oot:", and maps the 128 orange pages corresponding to the next 7 unknown bits just-in-time. Evicting the TLB, triggering the unmasked Spectre gadget, and reloading all 128 orange pages will reveal an L1 TLB hit at 0x243a746f6000, leaking the byte "$". Now, with knowledge of the 4 bytes "ot:$", the attacker can repeat the process and leak the next byte. Nonetheless, since this intuitive style of *forward* leaking requires a sparse (and hence costly-to-remap) reload buffer, SLAM slides (and leaks) *backward* (towards lower addresses) by default, as also shown in Figure 10 (bottom-left). This strategy yields a compact, fully paged reload buffer (yellow in figure) we can efficiently `mremap` at every iteration using a single system call. Care should be taken not to remap into our own code/data, but this can be easily accomplished by a linker script placing exploit code/data in a range of the user address space encoding at least one non-ASCII character. In practice, building a non-PIE binary is sufficient on Linux.

7.3. Summary

We have shown that, while masked Spectre gadgets normally produce 64-bit (noncanonical) secrets, we can enhance our TLB Evict+Reload covert channel with entropy-reducing techniques to leak such high-entropy secrets byte-by-byte, similar to classic masked Spectre gadgets.

C3 Solution. SLAM relies on backward sliding and just-in-time remapping to leak high-entropy secret pointers byte-by-byte with a movable, 128-page reload buffer. We need N bytes of known data for N-level paging. This strategy generalizes our 1-bit noncanonical user pointer disclosure primitive to *arbitrary ASCII string disclosure*.

8. End-to-End Covert Channel

Armed with our ASCII string disclosure primitive, we can now build an end-to-end SLAM covert channel in preparation for our end-to-end exploit. This is to demonstrate a kernel-resident sender can reliably transmit data to a userland receiver. For this purpose, we need a *protocol* between sender and receiver, a way to *evict* (only) L1 TLB and L1 cache, and a way to handle *noise*. We will later reuse some of these building blocks for our end-to-end exploit.

8.1. Protocol

We use a simple protocol with the sender transmitting ASCII characters by transiently executing an unmasked gadget such as the one in Figure 1 and the receiver retrieving them via TLB Evict+Reload. To transmit one character at the time, the sender slides backward and the receiver remaps a 128-page reload buffer just-in-time. For synchronization, we simply allow the receiver to send a *ready for next iteration* signal via a predetermined syscall. For error correction, we use the inherent redundancy in our TLB signal, with only 7 out 31 bits used for data—i.e., we require the remaining bits to match the known signature. Upon mismatch (error), the receiver requests a retransmission. We consider three covert channel (4-level paging) variants: (i) AMD TENA (natively bypassing canonicality checks on vulnerable microarchitectures); (ii) Intel LAM (simulated by sign-extending bit 47 of the secret pointer to bits 62:48, as described in the ISA [21]); (iii) AMD UAI (expanding LAM's sign extension to bit 63).

8.2. Evictions

Our signal relies on the small latency difference of L1 vs. L2 TLB hits, which we measure with the timestamp counter and a counting thread [32], [46], [47] on Intel and AMD (respectively). As explained in Section 5.3, to reliably measure such difference we need to evict L1 TLB and L1 cache, while preserving their L2 counterparts, before the transmission occurs. To this end, we use a single virtually *and* physically contiguous eviction buffer, as accessing virtually (physically) consecutive pages puts minimal pressure on the L2 TLB (cache). Meanwhile, by carefully choosing at what page offsets we access the eviction pages, we use the same memory accesses to simultaneously evict the L1 TLB and the first L1 cache set (used for reloading). We allocate our eviction buffer by mapping a 2 MB huge page and then splitting it into consecutive 4 KB pages with `mprotect`.

8.3. Noise

We use standard techniques to deal with noise such as *pointer chasing* to implement evictions [35] and repetitions to sample the signal. In details, we repeat every single gadget iteration $2 * 128 * R$ times. The 2 factor ensures a first gadget repetition speculatively loads the secret pointer in the L1 cache, so a subsequent repetition can access it quickly no matter how short the speculation window. The 128 factor is to distribute the reload step across 128 gadget repetitions (one per reload entry), a simple way to combat noise from the TLB prefetcher. Finally, R is the number of microarchitecture-specific repetitions (or simply *repetitions* hereafter) to tune the accuracy-performance trade-off.

8.4. Covert Channel Evaluation

To evaluate our three end-to-end covert channel variants, we set up the following experiment. The sender generates 64 KB of random ASCII data and appends it with a predetermined magic value known to the receiver. The sender (receiver) transmits (receives) the data backward, starting from the magic value. To combat noise, we use $R = 4$ repetitions on Intel and $R = 32$ repetitions on AMD. We

CPU	Canonicality Bypass	Average Bandwidth	Standard Deviation	Retrans. Rate
i9-13900K	Intel LAM	1.37 KB/s	15.7 B/s	0.0%
Ryzen 7 2700X	AMD TENA	41.5 B/s	27.6 B/s	2.7%
Ryzen 7 2700X	AMD UAI	47.6 B/s	25.5 B/s	0.1%

TABLE 1: End-to-end covert channel bandwidth, standard deviation, and retransmission rates for the different variants.

repeated our experiments 10 times and reported a number of statistics in Table 1. As shown in the table, the selected number of repetitions result in a high-accuracy channel, with all the characters successfully received with a $< 3\%$ retransmission rate. However, as evident from the difference in the number of repetitions, we experienced much more TLB noise on AMD than on Intel. This ultimately resulted in much lower bandwidth and fluctuations across runs on AMD. Moreover, we observed negligible bandwidth differences between the AMD (UAI and TENA) variants, evidencing (i) little impact from the added UAI masking and (ii) consistently successful TENAs. We conclude that, for all three canonicality bypass variants, we can build reliable end-to-end covert channels using unmasked gadgets, with the best performance/accuracy observed on Intel.

9. BHI Gadget Analysis

Our next step to demonstrate SLAM's capabilities is to mount end-to-end exploits. We specifically focus on the BHI Spectre variant [11], since previous work has hypothesized exploitation is only feasible with special features such as unprivileged eBPF (now disabled) [9], [14]. To this end, we developed a simple gadget scanner based on Angr [48] to find BHI-compliant unmasked gadgets in target software. Our scanner takes a binary and a candidate gadget entry point (i.e., indirect branch target for BHI) as input and symbolically executes the binary from the entry point onward, up to a maximum number of instructions or basic blocks.

We use symbolic *labels* to model attacker controllability and secret accesses. To detect generic (indirect branch-agnostic) BHI gadgets, we mark all registers with the *controlled* label at the entry point. We also mark any 64-bit value loaded via a *controlled* address with the *secret* label. We propagate labels throughout instructions when the result depends on labeled input. Finally, we detect valid secret pointer dereferences (hence gadgets) if the address of a load/store has the *secret* label. This strategy finds unmasked gadgets that contain at least a controlled load reading a 64-bit pointer that is later translated by another load/store.

9.1. Gadget Evaluation

We applied our gadget scanner to Linux kernel 6.3, with the maximum number of instructions per gadget set to 40. To collect the list of entry points, we used an LLVM pass created by Intel researchers [9]. The pass produced a total of 31,021 indirect branch targets in the kernel. From this list, our gadget scanner found as many as 61,665 secret translations across 16,046 targets—each representing an unmasked gadget with one or more secret translations. Of all gadgets, 4,194 are *simple*, i.e., consist of a single basic block and have a trivial data flow between secret pointer read and dereference (only additions allowed). As shown in Figure 11, most *simple* gadgets are found within a few instructions. However, other gadgets are still found after 40 instructions and > 20 basic blocks.

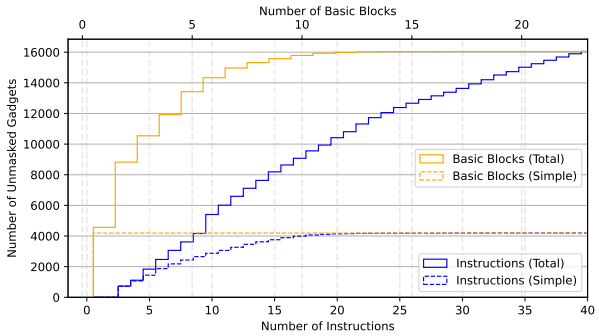

Figure 11: Length of the unmasked gadgets found by our scanner, in terms of instructions and basic blocks.

We also considered the characteristics of the individual gadgets, with results summarized in Table 2. As shown in the table, both store and load instructions contribute to secret translations, but loads are much more common, especially for *simple* gadgets. Moreover, we found gadgets matching arbitrary controllability requirements. In other words, as shown in the table, no matter which particular argument register(s) the attacker controls, they will find matching gadgets. As expected, the first (rdi) argument register (e.g., required to exploit the gadget in Figure 3) dominates, covering over 80% of the gadgets, but we found gadgets covering all the other argument registers even for the *simple* category.

Overall, our analysis revealed thousands of gadgets with flexible controllability requirements, even if we restrict ourselves to the short lengths of the *simple* category. Separating out statistics on *simple* gadgets is convenient, as such gadgets with straightforward data flow are a good approximation of the *practically exploitable* SLAM gadgets. In other words, modulo unexpected microarchitectural constraints, their exploitability is only subject to *reachability*—as the attacker needs to lure the kernel into executing the gadget before mounting a BHI attack. We validate this claim next.

9.2. Gadget Exploitability

To study the exploitability of *simple* gadgets, we first approximated "easy" reachability by means of kernel fuzzing. Specifically, we considered all the gadgets syzkaller [49] can reach within 24 hours. This led to a total of 1,808 gadgets, with 633 gadgets in the *simple* category. To assess whether the latter are practically exploitable, we manually analyzed randomly sampled gadgets matching the requirements of our end-to-end exploit (i.e., control over rdi, see later) and stopped after finding 10 exploitable gadgets.

Gadget	Translation	rdi	rsi	rdx	rcx	r8	r9	>1	all
Simple	load	3,313	738	69	18	3	2	2	4,105
Simple	store	257	41	4	1	0	0	0	301
Simple	all	3,388	753	72	18	3	2	2	4,194
Total	load	13,475	2,354	427	101	30	13	349	15,640
Total	store	3,864	718	116	16	10	4	88	4,734
Total	**all**	13,821	2,428	454	103	31	13	367	16,046

TABLE 2: Number of (simple) gadgets exploitable with control over different registers, by translation type (load/store).

Our analysis ultimately led to the manual inspection of 13 gadgets, revealing only 3 gadgets that are still effectively unreachable: 2 were only reachable by `root` and 1 was only sparingly reachable—a poor match for practical exploitation. We manually determined the other 10 gadgets to be exploitable, as inspection revealed no additional constraints. To further validate our analysis, we successfully integrated 5 of these gadgets in our end-to-end exploit chain, as detailed later. Overall, our analysis shows that reachable gadgets in the *simple* category are a good (under)approximation for practically exploitable ones and the attack surface is at *least* in the order of hundreds of gadgets for BHI alone.

10. End-to-End Exploit

Armed with exploitable BHI gadgets found by our gadget scanner, we now build an end-to-end SLAM exploit for upcoming Intel CPUs with LAM support and the latest Ubuntu (with eBPF disabled), ultimately leaking the root password hash from Linux' `/etc/shadow` file stored in kernel memory. At a high level, our exploit follows a workflow similar to our end-to-end covert channel, except the kernel no longer actively cooperates in the covert transmission. As result, our exploit needs to lure the *victim* kernel into speculatively executing a target gadget of interest using the Spectre BHI variant and disclose the secret data to an unprivileged user process controlled by the attacker.

10.1. Exploit Phases

Our end-to-end exploit consists of four phases. First, we break KASLR. Second, we find a Branch History Buffer (BHB) collision to speculatively hijack control flow to the target gadget, as done by BHI [11]. Third, we locate the `/etc/shadow` file in the kernel's physical memory mappings. Finally, we leak the ASCII content of the file and the root password hash.

Breaking KASLR. While there are many ways to break KASLR on commodity systems [50], our exploit relies on an oracle able to detect valid/invalid kernel memory addresses based on prefetch side channels [36]. We use the oracle to detect the (randomized) location of the Linux kernel's *direct map* of physical memory.

Finding BHB collisions. Building on prior [11] and concurrent [51] BHI work, our exploit finds BHB collisions to speculatively hijack the syscall dispatcher (our victim indirect branch) to a target gadget. Such indirect branch is particularly amenable to BHI exploitation [11], as (i) it is executed early on during the execution of a syscall and (ii) it provides the attacker with full control over (stack) memory referenced by the first argument register. The latter provides support for common unmasked gadgets that first load a (controlled) pointer via rdi, then use such pointer to read a 64-bit secret from an arbitrary kernel address, and then dereference that secret as a pointer (e.g., a 3-load pointer chase). To enlarge the transient execution window, we build L2 cache eviction sets [28] and evict from L2 the syscall table entry matching the attacker-issued syscall number (and referenced by the hijacked indirect branch).

To find BHB collisions and lure the indirect branch predictor into speculatively jumping to a target gadget, we use BHI's brute-force strategy [11], repeatedly generating a random branch history and issuing the victim syscall. This strategy requires an oracle to detect a successful collision, i.e., gadget speculatively reached by the victim syscall. For this purpose, we use a 2-step oracle. In Step 1, we pass a pointer into our reload buffer to the victim syscall and ultimately to the memory location that would be accessed by our gadget via rdi. If we reach the gadget, then the (second) load that would normally read the secret will instead attempt to read from our reload buffer, giving us a measurable TLB signal. This fast-path strategy is prone to false positives, as any mispredicted path performing a rdi-based pointer chase may inadvertently give us a signal. To address this problem, Step 2's slow-path strategy swaps our reload buffer pointer with a pointer to known kernel data, which the third gadget load will in turn attempt to dereference giving us a TLB signal. To inject known data into the kernel, we allocate a huge user page with a known magic value and repeatedly invoke Step 2 a sufficient number of times for the 2 MB-aligned kernel pointer to eventually land on the magic value somewhere in the kernel's direct map.

Locating the secret. Armed with a reliable BHI information disclosure primitive (i.e., via speculative control flow hijack to a target gadget), we need to locate the secret data, i.e., the `/etc/shadow` file. First, we make sure the data resides in physical memory by executing the `passwd` program. Next, we locate the shadow file by repeatedly effecting our primitive to scan physical memory via Linux' direct map for a 4 KB page that starts with "root:$".

Leaking the secret. The "root:$" prefix we used to locate the shadow file is unsuitable to kick-start our default byte-by-byte leakage strategy, as we leak backward. To address this problem, we simply start leaking from a known suffix. The suffix of choice is "deamon", the second user entry in the file. After locating the "deamon" signature, we leak backward byte-by-byte until we obtain the full root password hash. In case the exploit fails at any of the phases above (e.g., no BHB collision found), we simply restart from the first phase until we eventually leak the secret.

10.2. Exploit Evaluation

We evaluated our end-to-end SLAM exploit on Ubuntu 22.04 (Linux kernel 6.3) running on an Intel i9-13900K with 64 GB of RAM. As even such latest-generation Intel CPU

Gadget	Gadget size	Leakage rate	End-to-end run time
shmem_statfs	13 instr	345.4B/s	0.5m
sel_read_mls	14 instr	18.4B/s	0.6m
kernfs_seq_show	4 instr	376.9B/s	1.0m
raw_seq_start	8 instr	21.7B/s	1.7m
cgroup_seqfile_show	7 instr	363.0B/s	2.1m
kernfs_fop_read_iter	12 instr	56.3B/s	4.2m
proc_single_show	11 instr	6.3B/s	8.9m

TABLE 3: Exploitation results for different gadgets.

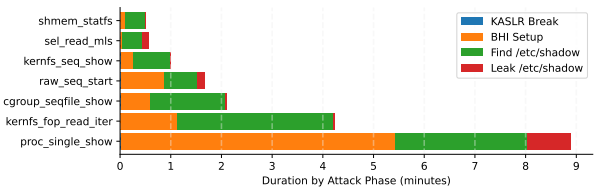

Figure 12: Exploit run time breakdown for different gadgets.

does not yet support LAM, we simulated LAM in software, as described in Section 8.

We instantiated our end-to-end SLAM exploit using 7 different unmasked gadgets: 5 from our manual analysis' random sample (cf. Section 9) and 2 more we had hand-picked earlier to develop SLAM. Figure 3 lists one of the gadgets we exploit. We ran each instance 10 times and report average statistics in Table 3. As shown in the table, our exploits do not appear bottlenecked by the speculation window size, with no correlation between size and leakage rates. Nonetheless, although all the 7 exploits leak the root password hash within minutes, we observed differences in leakage rates and end-to-end run times. In both dimensions, they boil down to the different gadgets being subject to different microarchitectural effects, e.g., aliasing. While it is hard to isolate all such effects, one factor that stood out for the "slowest" gadget (`proc_single_show`) was the unreliability of the BHB collisions ($< 60\%$ accuracy) compared to all the others ($> 95\%$ accuracy), harming leakage performance.

However, we note that, as we leak the root password hash (98 bytes), leakage rate variations affect the end-to-end run times only slightly. Figure 12 provides a more detailed breakdown. As shown in the figure, the run times are largely dominated by the time to find BHB collisions for BHI and the time to find the `/etc/shadow` file. In general, the BHI setup seems to impact the exploits the most, dominating their run time and with much larger fluctuations across exploit attempts compared to the other phases (stddev: $37\% - 114\%$ vs. $< 4\%$). Overall, our analysis shows end-to-end SLAM exploits are practical and can exploit a variety of unmasked gadgets on vulnerable microarchitectures.

11. Mitigations

To mitigate SLAM attacks, one can either hinder its covert channel or the exploitation of its gadgets.

11.1. Hindering the Covert Channel

To hinder SLAM's covert channel, the most natural solution is to extend SMAP semantics to address translation—i.e., preventing user address translation from the kernel. Although this strategy cannot stop SLAM-style exploits that target secret kernel pointers, such exploits are restricted to secrets that happen to encode *valid* kernel addresses (e.g., no ASCII strings). In short, their residual attack surface is significantly smaller. While this solution generally requires hardware extensions, one can repurpose an upcoming Intel feature, LASS [21], to implement the proposed semantics.

With LASS enabled, user and kernel execution can only access virtual addresses with their highest order bit set to 0 and 1 respectively. All the (speculative) accesses that violate this requirement are explicitly documented not to access any paging structures, including TLBs—thereby mitigating SLAM. LASS was originally designed to mitigate zero-day Meltdown-type attacks, with user execution attempting to perform a *secret access* on kernel memory. With SLAM, we show LASS plays an important role to mitigate Spectre-type attacks as well, in that it also hinders powerful translation *covert channels* other than the secret access.

LASS has however been added to Intel's ISA two years later than LAM [21]. On Intel platforms equipped with LAM but not LASS, as well as on AMD (which has not announced any LASS-like features), software-based alternatives are needed. A simple solution is to have the kernel disallow Intel LAM / AMD UAI features for unprivileged processes, restoring canonicality checks and hindering SLAM's non-canonical address translation covert channel. However, this simple option may limit the applicability of LAM / UAI in both mitigation and testing scenarios. A more principled mitigation is to let the kernel disable LAM / UAI on kernel entry and re-enable it when returning to userland (as needed), preserving canonicality checks in kernel execution.

To estimate lower-bound performance overhead of such mitigation, we ran the LMbench benchmark suite [52] and measured the cost of issuing (no-op) `CR3` and `EFER` register updates—as required by LAM and UAI (respectively)—on our Intel and AMD machines (respectively). Specifically, we measured the run-time overhead for Linux kernel 6.3 patched with the mitigation compared to the baseline. Table 4 presents our results. While some (syscall-less) benchmarks are unaffected (Pagefaults), the average overhead is nontrivial ($> 25\%$) and the worst-case (Simple syscall) is $> 3\times$. Although only processes using LAM / UAI are affected, the cost is significant for syscall-intensive programs. Furthermore, our results only provide a lower bound for the overheads, as upcoming CPUs would have to issue a meaningful (rather than no-op) update to the registers, other than software-masking user pointers in the kernel. Finally, such mitigation (similar to LASS) cannot stop any potential user-to-user attacks or protect (AMD) microarchitectures that are vulnerable to TENA [43], [44].

Benchmark	Intel i9-13900K		AMD Ryzen 7 2700X	
	Baseline	Overhead	Baseline	Overhead
Simple syscall	0.046μs	258.7%	0.078μs	213.2%
Simple read	0.069μs	194.6%	0.168μs	100.5%
Simple write	0.054μs	225.2%	0.121μs	140.7%
Simple stat	0.242μs	63.3%	0.573μs	30.1%
Simple fstat	0.172μs	75.9%	0.455μs	38.4%
Simple open/close	0.561μs	50.9%	1.297μs	26.5%
Select on 10 fd's	0.123μs	101.1%	0.344μs	48.8%
Select on 100 fd's	0.373μs	42.6%	1.158μs	14.8%
Select on 250 fd's	0.793μs	19.9%	2.474μs	7.1%
Select on 500 fd's	1.516μs	10.1%	4.754μs	4.1%
Select on 10 tcp fd's	0.135μs	95.7%	0.420μs	40.2%
Select on 100 tcp fd's	0.821μs	20.0%	4.068μs	4.4%
Select on 250 tcp fd's	1.988μs	7.4%	10.179μs	2.1%
Select on 500 tcp fd's	3.967μs	3.5%	20.442μs	1.0%
Signal handler install	0.078μs	163.5%	0.187μs	90.2%
Signal handler	0.527μs	25.6%	1.367μs	12.3%
Protection fault	0.263μs	4.1%	0.238μs	2.0%
Pipe latency	2.032μs	8.2%	5.926μs	5.8%
Unix socket stream	3.163μs	23.4%	5.278μs	20.8%
Process fork+exit	51.740μs	-1.9%	111.373μs	0.5%
Process fork+execve	136.171μs	2.6%	292.526μs	1.7%
Process fork+/bin/sh	297.737μs	2.7%	644.444μs	3.0%
Pagefaults	0.092μs	-1.6%	0.154μs	-7.8%
Geometric mean	-	**46.8%**	-	**27.5%**

TABLE 4: Run-time LMBench performance overhead of the Intel LAM- or AMD UAI-switching mitigation compared to the baseline on Linux kernel 6.3. We performed 11 repetitions per benchmark and report median results.

11.2. Hindering Gadget Exploitation

As SLAM can in principle abuse any Spectre variant (not just BHI), one can turn to generic Spectre gadget mitigations. A Spectre variant-agnostic solution is to annotate exploitable unmasked gadgets with fencing operations such as the `lfence` instruction or more efficient embodiments such as SLH [53]. However, as we showed, unmasked gadgets are pervasive in the kernel and precisely pinpointing all the practically exploitable ones is difficult. On the other hand, fencing *all* such (pointer-chasing) gadgets is likely to incur significant overheads [54].

For Spectre-v2/v5 attacks [11], [14], an option is to rely on hardware support to restrict indirect branch handling, as done in some upcoming processors [11]. On contemporary processors, researchers have also suggested the FineIBT mitigation (available only on Intel) [38] as a way to raise the bar for gadget exploitation. Specifically, FineIBT attempts to enforce fine-grained CFI in the speculative domain. Due to its low-latency CFI check, the net effect of the mitigation is to constraint gadget execution to a very short speculation window. Indeed, based on a minimalistic masked gadget filling a cache line (and no SMT contention to enlarge the speculation window [55]), the FineIBT authors measured a faint signal of only 17 out of 10M executions.

However, SLAM shows that unmasked gadgets can be exploited with a low-latency TLB covert channel (requiring only an L2 $\rightarrow$ L1 TLB fill, as opposed to a cache fill), reducing the speculation window size requirements. To evaluate FineIBT's effectiveness against SLAM, we inserted FineIBT-protected gadgets of different lengths (i.e., number of dependent loads including secret translation) into the kernel. By mistraining FineIBT's check and creating SMT contention [55] to delay its resolution, we set out to leak kernel data from a user process with SLAM. Figure 13 presents our 5-run results on Intel i9-13900K. Despite having observed noisy and system state-dependent behavior, our results show we can fit up to three dependent loads—the first two hitting the L1 cache and the last one hitting the L2 TLB—inside a FineIBT-protected speculation window with a reasonable signal. This is sufficient to exploit common SLAM gadgets such as the 7 used in our end-to-end exploit. Hence, our experiments show that, while FineIBT does raise the bar for exploitation, it is insufficient to mitigate SLAM.

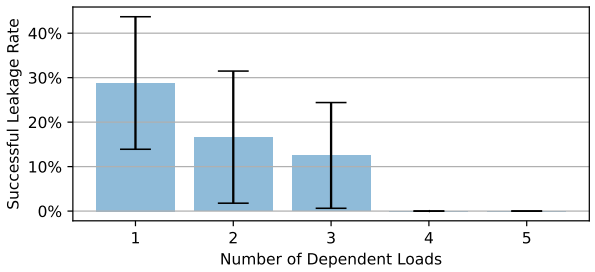

Figure 13: Successful leakage rate under mistrained FineIBT and SMT contention vs. number of dependent gadget loads.

12. Related Work

We briefly survey closely related work on MMU (microarchitectural) attacks and on Spectre covert channels.

MMU attacks. There is much prior work on exploiting the MMU's address translation for microarchitectural attacks. Some efforts target the MMU as a *confused deputy* to mount Rowhammer [35], [56] or cache attacks [57] on the attacker's behalf and bypass certain mitigations. Other efforts target the MMU as a *victim* of a side-channel attack to break ASLR [32], KASLR [36], [58], [59], [60], or purely for reverse engineering [34], [35], [40]. Yet other efforts target the MMU as an *attack vector* to mount covert- or side-channel attacks on victim software, abusing the TLB [34], [35] or the page table walks [61]. In contrast to these efforts, SLAM exploits the MMU to craft a Spectre covert channel.

Spectre covert channels. Since the original Spectre disclosure [1], there has been much work on studying Spectre covert channels, including classic array-based variations [1], [6], [9], [13], [14], [18], direct [62], [63], [64] or indirect [64] branches, MDS [16], [17], AVX instructions [65], and Rowhammer [66], [67]. All these covert channels (including existing TLB-based ones [6], [18]) operate with variations of *masked* Spectre gadgets, transmitting low-entropy secrets over microarchitectural state. In contrast to these efforts, SLAM exploits *unmasked* high-entropy gadgets for the first time, uncovering a new significant attack surface in modern kernel/hypervisors such as Linux.

13. Conclusion

We presented SLAM, a new Spectre attack based on a noncanonical address translation covert channel. To craft such covert channel in practice, SLAM relies on L1 TLB Evict+Reload combined with canonicality check bypassing and entropy-reducing techniques. Specifically, to bypass canonicality checks, SLAM can exploit hardware masking features in upcoming Intel/AMD CPUs or the TENA vulnerability in existing AMD CPUs. The impact of SLAM's new covert channel is significant, unlocking the vast attack surface of unmasked Spectre gadgets in modern kernels. Our experimental results show that such gadgets are abundant and, as a demonstration, we achieved a *Grand SLAM* of 7 end-to-end Intel/BHI exploit instances leaking the root password hash on the latest Linux kernel with eBPF disabled.

14. Disclosure

We disclosed SLAM to affected parties, specifically Intel, AMD, Arm, and Linux. Intel further notified other potentially affected software vendors. In response to SLAM, Intel made plans to provide software guidance prior to the future release of Intel processors which support LAM. Linux engineers developed patches to disable LAM by default until further guidance is available. ARM published an advisory [68] to provide guidance on future TBI-enabled CPUs. AMD did not implement guidance updates and pointed to existing Spectre v2 mitigations to address the SLAM exploit described in the paper.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback. This work was supported by Intel Corporation through the "Allocamelus" project, by the Dutch Research Council (NWO) through project "INTERSECT", and by the European Union's Horizon Europe programme under grant agreement No. 101120962 ("Rescale").

References

[1] 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. 1, 1, 2.3, 12

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: reading kernel memory from user space," in *USENIX Security*, 2018. 1

[3] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *USENIX Security*, 2018. 1

[4] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *IEEE S&P*, 2019. 1, 3, 7.1

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *CCS*, 2019. 1

[6] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *CCS*, 2019. 1, 1, 12

[7] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida, "Crosstalk: Speculative data leaks across cores are real," in *IEEE S&P*, 2021. 1

[8] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "Cacheout: Leaking data on Intel CPUs via cache evictions," in *IEEE S&P*, 2021. 1

[9] Intel, "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. 1, 1, 2.3, 9, 9.1, 12

[10] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *USENIX Security*, 2014. 1, 2.3

[11] 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. 1, 2.3, 3, 9, 10.1, 10.1, 11.2

[12] "Mitigating speculation side-channels in the Linux kernel," https://www.kernel.org/doc/Documentation/speculation.txt. 1

[13] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the Spectre era," in *CCS*, 2020. 1, 3, 5.1, 7.1, 12

[14] J. Wikner and K. Razavi, "Retbleed: Arbitrary speculative code execution with return instructions," in *USENIX Security*, 2022. 1, 3, 7.1, 9, 11.2, 12

[15] O. Kirzner and A. Morrison, "An analysis of speculative type confusion vulnerabilities in the wild," in *USENIX Security*, 2021. 1

[16] B. Johannesmeyer, J. Koschel, K. Razavi, H. Bos, and C. Giuffrida, "Kasper: scanning for generalized transient execution gadgets in the Linux kernel," in *NDSS*, 2022. 1, 12

[17] A. S. Jordy Zomer, "Finding gadgets for CPU side-channels with static analysis tools," https://github.com/google/security-research/blob/master/pocs/cpus/spectre-gadgets/README.md. 1, 12

[18] K. Loughlin, I. Neal, and J. Ma, "DOLMA: Securing speculation with the principle of transient non-observability," in *USENIX Security*, 2021. 1, 5, 12

[19] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: attacking ARM pointer authentication with speculative execution," in *ISCA*, 2022. 1

[20] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive Last-Level Caches," in *USENIX Security*, 2015. 1, 5.3

[21] Intel, *Intel® Architecture Instruction Set Extensions and Future Features Programming Reference*, 6 2023. 1, 2.2, 6.1, 8.1, 11.1

[22] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *USENIX Security*, 2014. 2.1, 5

[23] "Hardware-assisted AddressSanitizer design documentation," https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html. 2.2

[24] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: Buffer overflow checks without the checks," in *EuroSys*, 2018. 2.2

[25] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBOUNDS: Memory safety for shielded execution," in *EuroSys*, 2017. 2.2

[26] N. Burow, D. McKee, S. A. Carr, and M. Payer, "CUP: Comprehensive user-space protection for C/C++," in *AsiaCCS*, 2018. 2.2

[27] AMD, *AMD Upper Address Ignore*, 2023. 2.2

[28] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE S&P*, 2019. 4, 10.1

[29] 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. 4

[30] "SMAP effects on exploitation," https://github.com/google/security-research/security/advisories/GHSA-m7j5-797w-vmrh. 5

[31] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX Security*, 2019. 5

[32] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *NDSS*, 2017. 5, 5.1, 5.1, 8.2, 12

[33] "TLBs, paging-structure caches, and their invalidation," https://github.com/Nils-TUD/Escape/blob/master/doc. 5.2

[34] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *USENIX Security*, 2018. 5.2, 5.3, 12

[35] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *USENIX Security*, 2022. 5.3, 5.3, 8.3, 12

[36] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *CCS*, 2016. 5.3, 10.1, 12

[37] M. Lipp, D. Gruss, and M. Schwarz, "AMD prefetch attacks through power and time," in *USENIX Security*, 2022. 5.3

[38] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "FineIBT: Fine-grained control-flow enforcement with indirect branch tracking," *arXiv preprint arXiv:2303.16353*, 2023. 5.3, 11.2

[39] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos, "TLB; DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering," in *USENIX Security*, 2022. 5.3

[40] 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. 5.4, 12

[41] M. Larabel, "Intel linear address masking "LAM" merged into Linux 6.4," https://www.phoronix.com/news/Intel-LAM-Merged-Linux-6.4, 2023. 6.1

[42] A. C.-A. Series, *ARM Cortex-A Series: Programmer's Guide for ARMv8-A*, 2023. 6.2

[43] C. Musaev, Saidgani {and} Fetzer, "Transient execution of non-canonical accesses," https://arxiv.org/pdf/2108.10771.pdf, 2021. 6.2, 11.1

[44] AMD, "Transient execution of non-canonical accesses," https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1010.html. 6.2, 11.1

[45] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *IEEE S&P*, 2016. 7.1

[46] D. Kohlbrenner and H. Shacham, "Trusted browsers for uncertain times," in *USENIX Security*, 2016. 8.2

[47] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard, "Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript," in *Financial Crypto*, 2017. 8.2

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE S&P*, 2016. 9

[49] Google, "Syzkaller," https://github.com/google/syzkaller. 9.2

[50] "Kernel address space layout derandomization," https://github.com/bcoles/kasld. 10.1

[51] "Anonymized." 10.1

[52] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX ATC*, 1996. 11.1

[53] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: Taking speculative load hardening to the next level," in *USENIX Security*, 2023. 11.2

[54] M. Larabel, "The brutal performance impact from mitigating the LVI vulnerability," https://www.phoronix.com/review/lvi-attack-perf, 2020. 11.2

[55] A. Milburn, K. Sun, and H. Kawakami, "You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection," *arXiv preprint arXiv:2203.04277*, 2022. 11.2

[56] Z. Zhang, Y. Cheng, D. Liu, S. Nepal, Z. Wang, and Y. Yarom, "PTHammer: Cross-user-kernel-boundary Rowhammer through implicit accesses," in *MICRO*, 2020. 12

[57] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: Why stopping cache attacks in software is harder than you think," in *USENIX Security*, 2018. 12

[58] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi, "TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs," in *EuroS&P*, 2020. 12

[59] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *CCS*, 2016. 12

[60] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE S&P*, 2013. 12

[61] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *USENIX Security*, 2017. 12

[62] J. Fustos, M. Bechtel, and H. Yun, "SpectreRewind: Leaking secrets to past instructions," in *ASHES*, 2020. 12

[63] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019. 12

[64] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I see dead micro-ops: Leaking secrets via Intel/AMD micro-op caches," in *ISCA*, 2021. 12

[65] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *ESORICS*, 2019. 12

[66] 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. 12

[67] Y. Cohen, K. S. Tharayil, A. Haenel, D. Genkin, A. D. Keromytis, Y. Oren, and Y. Yarom, "Hammerscope: observing DRAM power consumption using Rowhammer," in *CCS*, 2022. 12

[68] ARM, "TLB channels, SLAM-like attacks, and transient translation of non-canonical addresses," https://developer.arm.com/Arm%20Security%20Center/TLB-Based%20Side%20Channel%20Attack. 14

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper deals with Spectre-like vulnerabilities. More specifically, the authors present a new (family of) attack(s), dubbed SLAM, which takes advantage of a new, upcoming feature in Intel and AMD CPUs, called LAM/UAI (linear address masking/upper address ignore), in order to turn unmasked Spectre gadgets into exploitable memory disclosure primitives. Unmasked Spectre gadgets typically result in non-canonical address translation, and they are currently considered a non-issue – from a security/exploitation perspective. The paper discusses how LAM/UAI, which is basically the equivalent of ARM's TBI (top-byte ignore) on Intel/AMD CPUs, can effectively act as a "mask" during pointer dereferencing, thereby facilitating the exploitation of unmasked Spectre disclose gadgets. LAM/UAI canonicalizes pointers and ignores privilege level checks – e.g., SMAP. Building on this observation, the paper further discusses ways of (ab)using (a) page-table translations and (b) the TLB for constructing a low-noise, low-latency covert channel for disclosing secret information. Lastly, the authors proceed with introducing two novel techniques, namely sliding and just-in-time reload buffer remapping, to facilitate the end-to-end exploitation of unmasked gadgets. The paper demonstrates SLAM by breaking KASLR and leaking the root password from kernel memory.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Identifies an Impactful Vulnerability
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

A.3. Reasons for Acceptance

1) The result(s) presented in this paper are both important and timely, as the exploitation of unmasked Spectre gadgets is currently considered a non-issue, and such gadgets are mostly ignored from analysis tools, hardening frameworks, etc.
2) The security analysis of LAM/UAI is novel and insightful, while the techniques for canonicalizing addresses using LAM/UAI and constructing a disclosure primitive by abusing MMU elements are advancing our current knowledge re: Spectre attacks and improve the state-of-the-art in terms of low-noise, low-latency covert channel construction.

A.4. Noteworthy Concerns

LAM/UAI was emulated in software and hence the end-to-end effectiveness experiment(s) performed used that software analogue. Nevertheless, Intel, AMD, and ARM have acknowledged that the issue is real.