# TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs

Jakob Koschel
Vrije Universiteit Amsterdam

Cristiano Giuffrida
Vrije Universiteit Amsterdam

Herbert Bos
Vrije Universiteit Amsterdam

Kaveh Razavi
ETH Zürich

***Abstract*—Kernel Address Space Layout Randomization (KASLR) has been repeatedly targeted by side-channel attacks that exploit a typical unified user/kernel address space organization to disclose randomized kernel addresses. The community has responded with kernel address space isolation techniques that separate user and kernel address spaces (and associated resources) to eradicate all existing side-channel attacks.**

**In this paper, we show that kernel address space isolation is insufficient to harden KASLR against practical side-channel attacks on modern tagged TLB architectures. While tagged TLBs have been praised for optimizing the performance of kernel address space isolation, we show that they also silently break its original security guarantees and open up opportunities for new derandomization attacks. As a concrete demonstration, we present TagBleed, a new side-channel attack that abuses tagged TLBs and residual translation information to break KASLR even in the face of state-of-the-art mitigations. TagBleed is practical and shows that implementing secure address space isolation requires deep partitioning of microarchitectural resources and a more generous performance budget than previously assumed.**

## 1. Introduction

Kernel-level Address Space Randomization (KASLR) is a first line of defense against adversaries that aim to exploit software vulnerabilities in the kernel for escalating their privilege. While KASLR raises the bar for attackers, previous work has shown many different possibilities for side-channel attacks that can easily bypass KASLR [1], [2], [3], [4]. These attacks exploit the unified kernel and user address spaces that is exposed through various microarchitectural resources.

To stop these attacks, recent mitigations propose separating kernel and user address spaces on these microarchitectural resources [5], [6]. While secure, these mitigation would be expensive without tagged Translation Lookaside Buffer (TLB) available on all modern Intel processors. Tagging the TLB significantly reduces the overhead of these mitigations by avoiding TLB flushes on every privilege switch which is now necessary. As a result, tagged TLB is praised for enabling deployment of such mitigations in practice [5], [7]. After the public disclosure of speculative execution attacks [8], [9], [10], [11], major operating systems such as Linux and Windows turned these mitigations on-by-default.

In this paper, we show that while separating kernel and user address spaces mitigates some of the specualative execution attacks, they do not fullfill their original goal of protecting against side-channel attacks on KASLR. Ironically, tagged TLB, the optimization that makes separating kernel and user address spaces efficient, re-enables the sharing of the TLB entries across kernel and user address spaces. Our proof-of-concept exploit, TagBleed, uses the new leakage introduced by this sharing to fully break KASLR in spite of these deployed mitigations.

**KASLR Attacks & Defenses.** Existing side-channel attacks against KASLR [1], [2], [3], [4] target shared microarchitectural resources to derive information about secret locations in the virtual address space where kernel memory resides. These attacks are possible due to *unified* kernel and user address spaces supported by the CPU for reasons of efficiency. For instance, a unified virtual address space between a user process and the kernel allows the user process to measure timing differences of a triggered CPU exception when accessing a kernel address to determine whether that address is mapped, breaking KASLR [2], [4]. Similar attacks are also possible without even triggering an exception: by measuring the execution time of the prefetch instruction for kernel addresses one can probe the existence of address translation data structures in the CPU's translation caches [3]. This unification of address spaces even extends to microarchitectural resources such as the Branch Target Buffer (BTB), making it possible for attackers to find out the virtual address space of branch targets in the kernel from user space [1].

To mitigate this class of attacks, recent proposals advocate for isolation of the kernel address space. This can be enforced by explicitly flushing microarchitectural resources such as the BTB on privilege switches [6] (or implicitly flushing them with hardware mitigations such as eIBRS [12]) and placing the kernel memory on a separate address space [6], [5]. This stops the attackers' ability for probing the existence of kernel addresses from a user process. On every privilege switch (e.g., due to a system call) the address space translation structures cached by the CPU in the TLB (or translation caches [13]) need to be flushed because of a different kernel address space. The TLB in recent Intel processors improves the performance of these mitigations with tagging. Every entry is tagged with the address space identifier and as a result, it is no longer necessary to flush the (tagged) TLB on every privilege switch. Due to the improved performance, tagged TLBs have been praised for making these mitigations practical for deployment [5].

**TagBleed.** Unfortunately, a tagged TLB is not a panacea and the gain in performance comes at a significant security cost. Tagging implicitly re-enables the sharing of the TLB between different address spaces. This allows an attacker

to probe addressing information in the TLB left by kernel execution. As we will show, the leakage surface of tagged TLB is limited compared to known attacks against KASLR [1], [2], [3], [4]. Nevertheless, we show that this leakage is enough to fully derandomize KASLR when used in combination with a secondary side-channel attack that uses the kernel as a confused deputy [14] to leak additional information about its address space. Mounting these attacks is not trivial since kernel memory is mapped with huge and super pages (i.e., 2 MB and 1 GB) and (tagged) TLBs use previously-unexplored hashing functions for storing the translation information for these pages. Our proof-of-concept exploit, TagBleed, uses the information we obtained through reverse engineering to break KASLR in under one second using the aforementioned attacks despite all existing state-of-the-art mitigations.

**Contributions.** In summary, our contributions are:

- We highlight the security implications of tagging (previously-untagged) cache components for the first time. While tagging components improves performance, they can come at a security cost.
- We present an extended analysis of the architecture of the TLB in modern processors. Expanding on the existing knowledge of the TLB for normal pages [15], we reverse engineered the TLB architecture for both 2 MB huge pages and 1 GB super pages used when mapping kernel entries.
- The design and implementation of our practical attack, TagBleed, which derandomizes KASLR on a recent Linux system with current defense mechanisms deployed in under one second. TagBleed targets tagged TLB in combination with a confused deputy attack to fully break KASLR.

## 2. Background

**Virtual Memory.** In modern operating systems, each process has access to a private virtual address space and operates solely on virtual addresses. The translation to the actual physical addresses is the responsibility of the MMU (Memory Management Unit) which walks the multi-level page table structures that contain the virtual-to-physical mappings for each address space, as well as permission flags (such as the supervisor bit that indicates a page can be accessed from user space). Since these address translations are expensive and happen at each memory access, the MMU utilizes the TLB (Translation lookaside buffer) to cache the last few lookups—speeding up subsequent accesses to the same page by orders of magnitude.

In the operating systems with unified kernel and user address spaces that were popular until very recently, the kernel did not have an address space of its own, but rather shared the page tables of the running user process and relied on the supervisor bit to protect its pages from illegitimate accesses from the user process. As neither the address space (page tables) nor the content of the TLB needed to change on kernel boundary crossings, such an optimized memory organization was very efficient. The optimization is especially effective on processors without tagged TLBs, since they must perform a full TLB flush on every address space switch [16].

As a result of a barrage of side channel attacks [2], [4], [3], [8] that all abused the unified address space, modern operating systems recently abandoned it altogether and now provide the kernel with its own, completely separate address space. On Linux, this is known as KPTI (Kernel Page Table Isolation) [17], while Windows refers to it as KVA (Kernel Virtual Address) Shadow [18]. Fortunately, while the separation of address spaces is still quite expensive on older processors because of the TLB flushes, newer CPUs offer tagged TLBs where each entry contains an identifier (or "tag") of the address that owns it. For instance, Intel's x86_64 processors have supported tags in the form of 12-bit PCIDs (Process Context Identifiers) since the SandyBridge architecture [19]. When performing a lookup in the TLB, the entry's tag must match that of the currently active address space. Thus, flushes are no longer needed, as there is never any confusion about the validity of a TLB entry—greatly improving performance.

**Address Space Layout Randomization.** To prevent attackers from locating suitable targets to divert a program's control flow in the presence of a vulnerability, all major operating systems today support Address Space Layout Randomization (ASLR). ASLR randomizes the locations of the code, heap and stack in memory and forms an effective first line of defense against memory error attacks. The kernel variant of ASLR, known as KASLR, is deployed on all major operating systems. This requires attackers to first break KASLR as a crucial step in kernel exploitation.

KASLR randomizes the location of the kernel and drivers running in kernel space either at boot time or at driver load time. Since brute forcing the randomization in the kernel is typically not an option due to the high rate of kernel panics, the randomization entropy in the kernel can be lower than in user processes. For instance, at the time of writing, the entropy for the kernel image in Linux is 9 bits [20] , while the entropy for user space code is as high as 30 bits [21].

More specifically, KASLR in Linux randomizes different kernel regions differently. At boot time the kernel image is unpacked and relocated to a random location. Regions for the identity map, vmalloc and vmemmap are randomized separately. Finally, kernel modules are randomized the first time a module is loaded. As shown in Table 1, the kernel image has 9 bits of entropy, and kernel modules have 10 bits. Identity map, vmalloc and vmemmap are randomized with a shared entropy depending on the size of physical memory.

**TLB Tagging.** Switching address spaces on processors prior to TLB tagging is a costly operation due to the invalidation of all entries. Therefore tagging TLB entries with an identifier for its current address spaces was introduced as an optimization. Intel processors use the first 12 bits of the CR3 register to store a so called PCID (process-context identifier) [19]. Entries in the TLB will only be taken into account if the current PCID in CR3 matches the PCID of the entry. This makes context switching more performant, since the TLB does not require any flushing.

**Cache Attacks.** On address lookups, the TLB caches virtual to physical memory translations and as a shared resource between processes running on the same core, clearly introduces a side-channel risk that was exploited

TABLE 1. KASLR ENTROPY IN LINUX 4.19.4 FOR THE KERNEL IMAGE, KERNEL MODULES AND PAGE_OFFSET, VMMALLOC AND VMEMMAP. THE NUMBER OF POSSIBLE SLOTS FOR THE KERNEL IMAGE ARE DEPENDENT ON THE SIZE OF THE KERNEL IMAGE. THE ENTROPY AND END ADDRESS FOR PAGE_OFFSET DEPENDS ON WHETHER FIVE PAGE TABLE LEVELS ARE SUPPORTED AND HOW MUCH PHYSICAL MEMORY IS AVAILABLE.

| | start address | end address | entropy | possible slots | alignment |
|---|---|---|---|---|---|
| kernel image | 0xffffffff81000000 | 0xffffffffbe000000 | 9 | 488 | 2MB |
| kernel modules | 0xffffffffc0001000 | 0xffffffffc0400000 | 10 | 1024 | 4KB |
| page_offset_base, vmalloc_base, vmemmap_base | 0xffff888000000000 | 0xfffffe0000000000 | 15* | 25600* | 1GB |

* depends on amount of physical memory and CONFIG_RANDOMIZE_MEMORY_PHYSICAL_PADDING configuration.

in the TLBleed attack by Gras et al. [15]. However, the TLB is just one of many shared resources that have been used for side-channel attacks. Modern Intel CPUs have multiple levels of caches to speed up memory accesses. Specifically, each core has its own L1 and L2 caches and shares the last level cache (LLC) with the other cores. Since attacker and victim don't even need to run on the same core, the LLC is a particularly interesting target for a side channel attack [22], [23], [24], [25], [26]. Attackers can simply populate cache sets and then measure whether the victim process evicts their data. With this information the attacker can infer that the victim used addresses that map to the same cache set and researchers have shown that this is enough to leak sensitive information such as cryptographic keys [27], [28], [29], [30], [31].

**AnC.** The AnC [32] attack measures the timing of accesses performed by the MMU during virtual address translation to break ASLR from within the browser. Whenever an address translation is not cached in the TLB, the MMU does a page table walk. It reads offsets within the multi-level page tables in order to first dereference the next page table and then the address of the physical page. In order to speed up expensive page table walks, accessed parts of the page tables are cached within the CPU's data caches. This makes consecutive translations faster even if the MMU has to do a page table walk. Inevitably, caching parts of the page table leaves traces in the CPU cache depending on the translated virtual address. By partially flushing the data caches, AnC can measure which cache lines within the page table pages have been used during a translation. This information already significantly compromises the ASLR entropy. To fully break ASLR, AnC needs to find out which page table entries within the target cache lines are accesses by the MMU. This is achieved by accessing a large virtual contiguous buffer, and observing the transitions between activated cache lines, known as *sliding*. The demonstrated attack requires large accessible virtually consecutive buffers (e.g., 8 GB). While this is possible in the browsers, it becomes challenging when applied to the kernel.

## 3. Threat Model

We assume an attacker that can execute unprivileged code on the target system with a kernel that is hardened with all common mitigations, including KASLR and DEP [33]. For the hardware, we assume a modern processor with a tagged TLB. In this paper we focus on Intel processors with PCID-based TLB tagging, ARM and AMD however also provides a similar feature through ASIDs. The attacker aims to elevate privileges to ring 0 by exploiting a memory corruption vulnerability in the kernel or a kernel module. To do so, the attacker first needs to break KASLR. In general, breaking KASLR is possible via a software-based information disclosure vulnerability or a side-channel attack. We assume that the kernel does not contain a known information disclosure vulnerability and that powerful mitigations against side-channel attacks on KASLR are turned on [5], [6]. The attacker's aim is to bypass these mitigations and successfully break KASLR with a side-channel attack. In this paper, we mostly focus on the Linux kernel, but the techniques we develop will likely be applicable in other kernels as well.

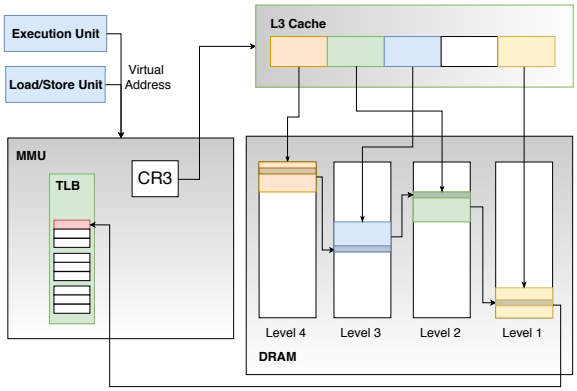


Figure 1. High level overview: On a memory access the MMU will translate a virtual address to a physical address using the page table structures. The result of the translation will be cached in the TLB and parts of the page tables in the L3 cache (LLC). By observing the state of the TLB and LLC we locate the position of kernel translations in the cache. Because the position is dependent on the translated virtual address we can successfully derandomize KASLR.

## 4. Attack Overview

Existing side-channel attacks on KASLR rely on the kernel being mapped in the same address space as the user process [3], [4], [2]. Specifically, these attacks can detect whether the kernel is mapped at a given address without the permission to *access* that address. By removing the kernel from the user address space and moving it to its own memory space these side channels are no longer possible [5], [6]. As kernel address space isolation ensures that kernel memory is only mapped while executing in kernel space, an attacker needs to perform a confused deputy attack [14] on the kernel, tricking it into performing the attack on itself.

Unfortunately for the attacker, the confused deputy attack is not possible with any of the existing techniques. For instance, the TSX attack [4] would require the kernel to access a user-controlled pointer in a hardware transaction. However, since the Linux kernel does not use transactions, this is not a feasible attack vector. The same is true for the `prefetch` instruction [3], while triggering invalid kernel page faults when performing the attacks described by Hund et al. [2] would crash the kernel. Therefore, our confused deputy attack should find other mechanisms. The simplest operation that we can force a kernel to perform is an access to its own memory. Memory accesses occur on every single instruction as the processor loads text and often data from memory. An important research question is whether it is possible to break KASLR using uncontrolled valid memory accesses performed by the kernel—and as we will see, the answer is yes.

Figure 1 shows how the information that the kernel maintains for address translation leaves traces in various caches when performing a memory access. On each memory access, the MMU performs a virtual to physical address translation, first consulting the TLB to see whether the result is already cached there. If the translation is not in the TLB, the MMU performs a page table walk to translate the virtual address into a physical one. Accessing page table entries in the page table results in caching parts of the page table in the CPU's data caches. Bits of the virtual address determine the set in the TLB and which part of the page table is being cached.

Ironically, the introduction of tagged TLBs, so important for the performance of systems with an isolated kernel address space, undermines the very isolation it should be helping, since it is now possible for a user process to probe TLB sets in its own address space to detect kernel activity in these same sets. Moreover the sets in which there is activity reveal information about the virtual address of kernel memory. However, breaking KASLR by observing such TLB activations presents several challenges:

**C1** Since the kernel is mapped using huge or super pages, we need to understand the TLB architecture for these page types and their addressing function. The TLB architecture for 4 KB pages has only recently been reverse engineered by Gras et al. [15], but the TLB behaviour for larger pages remains entirely unknown.

**C2** Given the small number of TLB sets, the partial information retrieved from a tagged TLB is unlikely to be sufficient to fully derandomize KASLR. We therefore need an auxiliary side channel to combine it with the side channel over tagged TLB. This raises the challenge of combining these side channels.

**C3** Timing a kernel memory access in a confused deputy-style attack requires forcing the kernel to perform a certain operation on behalf of a user process (e.g., a system call). Compared to existing attacks that simply time a single memory access by itself, the system call introduces significant additional noise. Building a practical side-channel attack in such a setting is challenging.

We first address **C1** by reverse engineering the TLB architecture for huge and super pages in Section 5. In Section 6, we then show how an EVICT+TIME attack on a tagged TLB combined with a constrained variant of the AnC attack [32] allows breaking KASLR (addressing **C2**). We further show how selective TLB flushing, made possible through our reverse engineering efforts, can significantly reduce noise to make our attack practical (addressing **C3**).

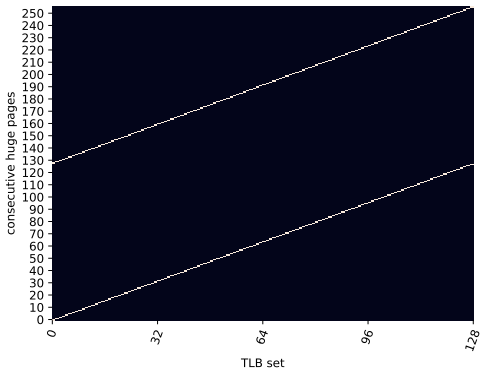


Figure 2. The graph shows the TLB set for 256 virtual consecutive huge pages starting at address `0x6ffe00000000`. The hash function for huge pages is linear, given by the 7 bits after the page offset.

## 5. Reverse Engineering the TLB

For the TLBleed attack [15], the authors present their efforts in reverse engineering the TLB architecture to understand how virtual addresses map to different sets in the TLB. They use Intel Performance Counters (PMCs) to gather fine-grained information about TLB misses and their level. Intel provides performance events like `dtlb_load_misses.stlb_hit` and `dtlb_load_misses.miss_causes_a_walk` which allow differentiating between a L2 sTLB hit and a miss.

With their findings they define the L1 TLB as linearly-mapped TLBs. This describes the hash function determining the TLB set of a virtual address. In a linearly-mapped TLB the hash function is given by $tlb\_set(va) = page_{va} \bmod s$. A TLBs architecture is defined by its sets *s*, ways *w* and its hash function *tlb_set(va)*. The L2 sTLB however is a complex-mapped TLB in recent Intel architectures. This means the hash function is not linear. In the Skylake sTLB, for example, the hash function is given by a *XOR-7* function which xors bits 19 to 13 with the bits 26 to 20 of the virtual address.

We first tested whether we can evict a TLB entry for a huge page (i.e., 2 MB) by flushing the whole TLB with 4 KB pages. Using performance counters we verified that we were able to evict the TLB entry of the 2 MB page from the L1 dTLB and the L2 sTLB. We conclude from this that both L1 dTLB and L2 sTLB are shared between 4 KB and 2 MB pages. This information already gives us the set and ways for the L2 sTLB since it is the same for normal pages. Contradictory to the information from `cpuid`, the L1 dTLB (2 MB pages) on Skylake therefore has 64 entries with 16 sets and 4 ways instead of the stated 32 entries and is shared with the L1 dTLB for 4KB page translations. Based on our findings the dTLB for 2MB pages, as well as the sTLB for 1GB pages are linearly mapped with the bits after the page offset. We,

TABLE 2. TLB ARCHITECTURE FOR SKYLAKE ARCHITECTURE BASED ON OUR REVERSE ENGINEERING EFFORTS. THE INDEXING COLUMN DETERMINES THE BITS USED FOR THE LINEAR INDEXING FUNCTION. ONLY THE STLB FOR 4 KB PAGES IS INDEXED WITH THE COMPLEX XOR-7 INDEXING FUNCTION.

| | L1 dTLB | | | | L2 sTLB | | | |
|---|---|---|---|---|---|---|---|---|
| | sets | ways | indexing | shared | sets | ways | indexing | shared |
| 4 KB page [15] | 16 | 4 | VA[15:12] | ✓ | 128 | 12 | XOR-7(VA[25:12]) | ✓ |
| 2 MB page | 16 | 4 | VA[24:21] | ✓ | 128 | 12 | VA[27:21] | ✓ |
| 1 GB page | - | 4 | fully associative | ✗ | 4 | 4 | VA[31:30] | ✗ |

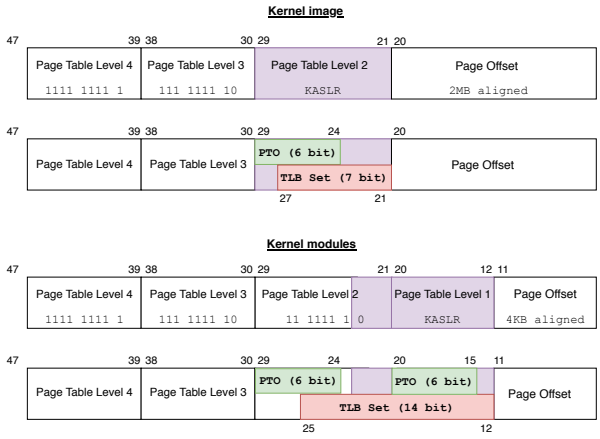


Figure 3. For the kernel image the whole second page table level (PTL) is randomized by KASLR. Using TLB sets we are able to randomize the lower 7 bits. Combining it with page table offset (PTO) information, which derandomizes the higher 6 bits, we can successfully break KASLR. For kernel modules PTL1 and the lowest bit of PTL2 are randomized. Using the TLB set, given by the XOR-7 hash function, together with the offset within the page table is enough to derandomize those 10 bits of entropy.

however, still need to reverse engineer the hash function that determines in which of the 128 sTLB sets a huge page is put in. We know that the sTLB is shared with 4KB pages where, on Skylake, the set is determined by a complex *XOR-7* hash function. To observe the addressing function for 2MB pages we evict one set in the sTLB at a time using 4KB pages. That allows us to observe which TLB set the huge page is mapped to. Figure 2 presents our results on Skylake for measuring the TLB set for 2MB pages. Based on our measurements, the sTLB set for 2 MB huge pages is determined by bits 27 to 21 of the virtual address (VA[27:21]). Contradictory to the complex *XOR-7* hash function for 4 KB pages, for 2 MB pages it is only addressed with a linear addressing function. An overview of our findings on the TLB architecture is summarized in Table 2.

## 6. TagBleed

In this section, we discuss the building blocks of our TagBleed attack. We first discuss how we can force the kernel to access its virtual memory to start off our confused deputy attack. We then describe how we utilized the knowledge we gained through our reverse engineering for leaking kernel virtual address information through the tagged TLB and to reduce noise during the attack. After that, we discuss how we utilized a constrained version of the AnC attack to break the remaining residual (2 bits) of entropy for the kernel text. We further extend our attack to derandomize the location of kernel modules and data.

### 6.1. Forcing Kernel Memory Access

Measuring a memory translation in the kernel is challenging. Simply measuring a single translation is not possible. We need to start the timer prior to entering the kernel and stop it once returned to user space. Measuring the entire kernel operation introduces additional noise caused by other instructions being executed. System calls are an easy way for a user process to communicate with the kernel. To minimize noise of other code executed during the system call, we took the shortest possible system call by providing an invalid system call number. Early in the system call handler the kernel will look for a system call with the provided number and abort because of the invalid argument. We note that the `SYSCALL` instruction is also considerably faster than `INT 0x80`, in order to shorten the execution path.

### 6.2. Leaking Through Tagged TLB

As mentioned in Section 2, KASLR support in Linux aligns the kernel text to 2 MB and randomizes bit 21 to 29 of the virtual address, in other words, the slot in second page table level (i.e., PTL2) as shown in Figure 3. We craft an EVICT+TIME attack, evicting one TLB set at a time and timing a target (kernel) memory access. If the TLB set with the desired entry is evicted, the MMU is forced to perform a page table walk. A page table walk is considerably slower than just using the cached translation from the TLB. Note that due to KPTI the TLB would be flushed entirely on a context switch if no TLB tagging would be in-place. With the tagged TLB, we now have the capability to selectively evict parts of the TLB and observe the effects across context switches. As already presented, the L2 sTLB set for huge pages is determined by VA[27:21], which allows derandomizing the lower 7 bits of the second page table level. The last 2 bits of PTL2 , which remain unknown, are derandomized by combining this attack with information obtained through page table walker discussed in the next section.

**Reducing noise.** Selectively flushing one TLB set at a time also reduces the noise. Not evicting the entire TLB massively reduces the amount of unwanted page table walks which create a large amount of false positives.

### 6.3. Confused Deputy Attack with AnC

The AnC attack on the MMU can break ASLR when the attacker can freely *slide* in the virtual address space [32]. AnC relies on the MMU caching parts of

```
for each tlb_set do
    for each cache_line do
        evict_l1_tlb()
        evict_l2_tlb_set(tlb_set)
        evict_cache_line(page_table_cache_line)
        past ← rdtscp()
        syscall
        now ← rdtscp()
        timing[tlb_set][cache_line] ← now − past
    end for
end for
```

Figure 4. Timing a system call by only evicting one TLB set and one page table cache line at a time.

the page table page on a page table walk. Depending on the virtual address, different parts of the page table end up being cached in the LLC. Using EVICT+TIME, AnC locates the cache lines containing the accessed page table entries by the MMU. However this will not reveal the complete virtual address since multiple 8-byte page table entries are stored in the same 64-byte cache line. AnC addressed this problem by accessing large virtually contiguous memory addresses, i.e., sliding. Accessing subsequent virtual addresses cause subsequent cache lines to be accessed by the MMU. The point at which a new cache line is accessed reveals the offset of page table entries in a cache line – fully derandomizing ASLR. While powerful, it is not trivial to apply the AnC attack to the kernel due to two reasons. First, we cannot make the kernel slide its address space, and second, each step of the AnC attack causes up to four cache line activations due to four levels of the page table, introducing false positives. We present a variant of this attack integrated into our tagged TLB side channel to leak the remaining 2 bits of entropy and making TagBleed more noise-resistant.

**Sliding.** As shown in Figure 3, the residual 2-bits of entropy left from our TLB attack are not related to the offset of page table entries within the cache lines. This means that we do not need to perform sliding to retrieve these two bits. As a result, *a single memory access* by the kernel is enough to break KASLR when combining these two side channels.

**Other PTLs.** In order to speed up page table walks Intel not only caches complete virtual to physical translations in the TLB, but also partial translations in its *translation caches* [13]. These caches allow the MMU to skip page levels during the translation. We make use these translation caches to force the MMU to skip page tables that are not interesting for KASLR (see Figure 3). This allows us to avoid false positives caused by other page table levels.

**Combining the side channels.** Figure 4 shows the high level operation of TagBleed when combining the tagged TLB and AnC attacks. We generate a two dimensional matrix with all combinations of evicting one TLB set and one cache line. Then we use a simple script to identify the best candidate. We first identify the best candidate for the TLB set. Since this derandomizes the lower 7 bits of PTL2, only the 2 highest bits are missing. Therefore we only need to choose between 4 possible cache lines.

TABLE 3. MICROARCHITECTURES USED IN EVALUATION.

| Vendor | Microarchitecture | CPU model | Year | TLB tagging |
|---|---|---|---|---|
| Intel | Haswell | i7-4650U | 2013 | PCID |
| Intel | Skylake | i7-6700K | 2015 | PCID |

Since the AnC attack gives us the upper 6 bits of PTL2, we can use the upper 4 bits of the best candidate from the previous step as a noise filter when selecting the final candidate.

### 6.4. Derandomizing Kernel Modules

Kernel modules are loaded with an offset randomized by KASLR when the first module is loaded in the system. In order to observe the signal, we need to force an access to a memory location within the kernel module. We achieve this by performing an *ioctl* call to a loaded kernel module. Kernel modules are mapped with 4 KB pages, contrary to 2 MB pages used for the kernel image. Therefore, as shown in Figure 4, bits 12 to 21 of the virtual address are randomized, since kernel modules are not 2 MB, but 4 KB aligned. This slightly changes our approach since the TLB indexing function for 4 KB pages is different than for 2 MB pages. For example, on Skylake the TLB set for 4 KB pages is determined by an XOR of bits 12 to 18 with the bits 19 to 25 of the virtual address. Hence, using TLB sets alone we cannot break any KASLR bits. The AnC attack, however, provides us with bit 15-20 through the offset of the activated PTL1 cacheline. Bits 19 and 20 allow us to find bits 12 and 13 as well, since bits 19 and 20 are XORed with bits 12 and 13 in the TLB's XOR-7 pattern [15]. The remaining entropy will be a single bit, since we cannot break KASLR at bit 14 and 21 while we know their XOR value.

### 6.5. Derandomizing Physmap

Derandomizing physmap is challenging because it is 1 GB aligned and randomized in PTL3 and PTL4. The TLB indexing function for 2 MB pages does not use those upper bits. Most of the physmap, however, is mapped with 1 GB pages with a separate TLB with its own indexing function. But since the sTLB for 1GB pages only has 4 sets, we can only use it to derandomize the lower 2 bits of PTL3. To derandomize the rest, we can make use of AnC to derandomize the higher 6 bits of both PTL3 and PTL4 reducing the entropy by another 10 bits. This still leaves us 4 bits of entropy (16 possible locations).

## 7. Evaluation

We evaluated TagBleed on a machine running Ubuntu 18.04 LTS (Linux kernel v4.19.4) with an Intel Core i7-4650U @1.70 GHz (Haswell) and 8 GB of RAM. In order to ensure portability across different architectures, we confirmed our evaluation results on another workstation running Ubuntu 18.04.1 LTS (Linux kernel v4.15.0) with an Intel Core i7-6700K @4.00 GHz (Skylake) and 16 GB of RAM. This also allowed us to confirm that a range of different TLB architectures is susceptible to our TagBleed attack. Table 3 details the CPUs and microarchitectures considered in our evaluation. In our evaluation,

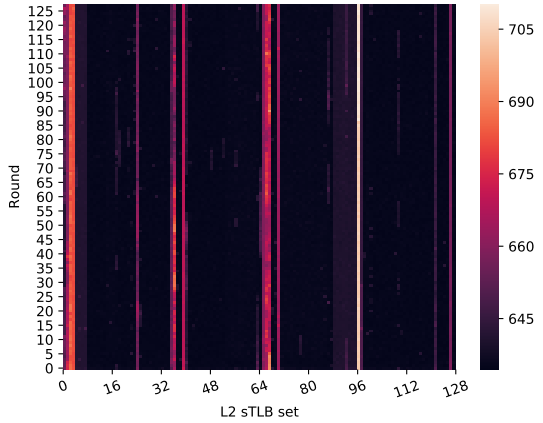


Figure 5. We can clearly see that evicting some TLB sets will slow down the time of an invalid system call. This can only happen when the (tagged) TLB is not fully flushed as the kernel switches address spaces with KPTI enabled.

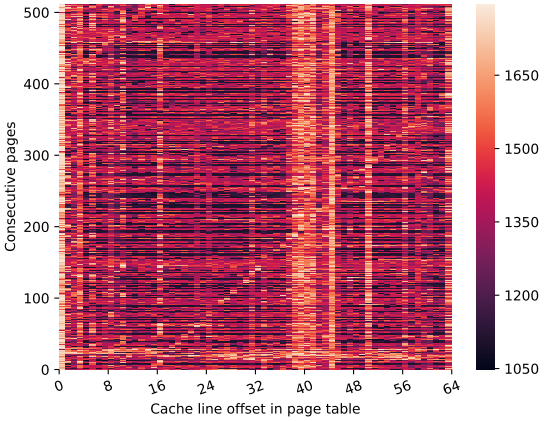


Figure 6. Although more noisy than a same-process page table walk signal, the sliding is still visible when timing the execution of an `ioctl` syscall to our kernel module. The offset within the buffer, which the kernel module accesses, is passed as an `ioctl` argument.

we targeted the derandomization of KASLR for the kernel image.

## 7.1. Side channel by TLB set eviction

We first evaluated our assumption on whether the partial TLB set eviction from user space can influence the MMU's virtual-to-physical memory address translation. Without KPTI, the (unified user/kernel) address space is not switched on kernel entry, so no TLB flushing is performed. With KPTI, however, the kernel updates the CR3 register to switch to the separate kernel address space. This operation does flush the TLB on legacy architectures, hindering our partial TLB set eviction strategy. However, on modern tagged TLB architectures, tagged entries are no longer flushed at mode switching time and we should be able to surgically trigger a page table walk only when evicting the correct TLB set.

Figure 5 validates our assumption, depicting the impact of evicting different TLB sets on the execution time of a dummy (i.e., invalid) syscall. Note that using an invalid syscall, that is a syscall with an invalid syscall number, is a convenient way to trigger short-lived kernel activity, but using any other short-lived syscall (e.g., `reboot` without root privileges) would also serve our purposes.

As Figure 5 shows, the execution time increases only when evicting specific TLB sets (revealing the virtual memory activity of the syscall). The signal persists if we disable KPTI, since the address space is not changed on a context switch, keeping the current state of the TLB. KPTI on a legacy TLB architecture without a tagged TLB requires a full TLB flush on a context switch. This clears the state of the TLB cache and therefore stops TagBleed, but it comes at a high performance cost for each user to kernel transition.

## 7.2. Side channel by cache line eviction

Our second assumption is that we can observe the cache lines being accessed during a kernel page table

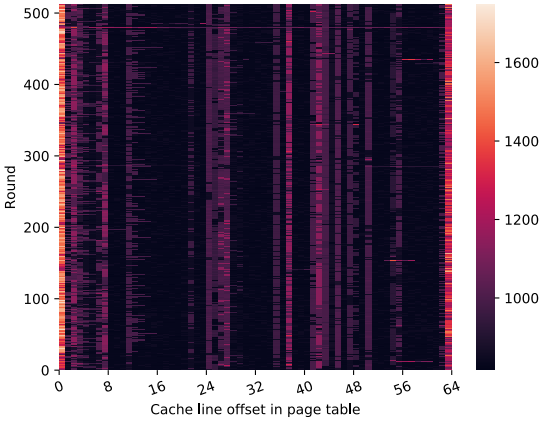


Figure 7. Cache evictions can slow down syscall execution in many unpredictable ways. As shown in the figure, not one but many different cache lines introduce cache misses in kernel execution even for an invalid syscall. The graph was created with the kernel target page being mapped using the sixth cache line, whose signal does not even stand out compared to other cache lines.

walk. In order to test this assumption, we built a kernel module to perform an AnC-style cache attack to monitor the page table walks performed by the MMU. The kernel module gives us the ability to access a given offset within a kernel buffer to perform sliding on the virtual address space and make the signal more visible. By accessing virtually contigous pages, which we define as sliding, the cache line of the page table entry will also be incremented. We then time the execution of an `ioctl` syscall that causes the kernel module to access a byte at a given offset. Figure 6 validates our assumption, depicting the signal for 512 contiguous virtual memory pages measured by timing the execution of the `ioctl` syscall.

However, when we repeat the experiment with an invalid syscall and without our kernel module (and therefore without sliding), the signal becomes very noisy, as depicted in Figure 7. This shows that, due to the entire cache activity of the kernel being impacted by cache

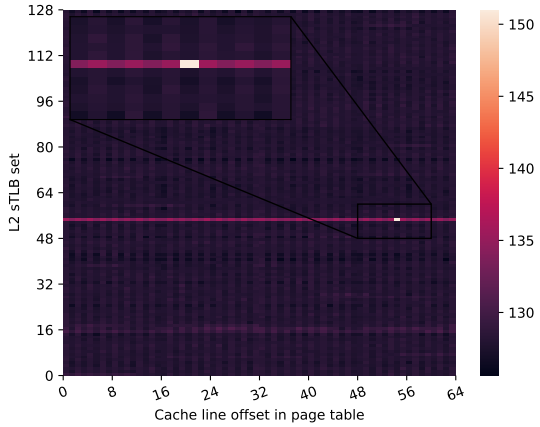


Figure 8. A combined EVICT+TIME attack on a user-level huge page access. Only when evicting the TLB set 55, the MMU performs an expensive page table walk. Moreover, when evicting cache line 54, the page table entries need to be loaded from memory which slows down the page table walk.

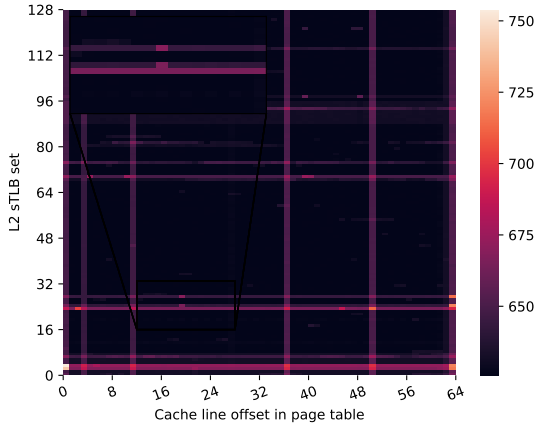


Figure 9. A combined EVICT+TIME attack on an invalid syscall. Based on our solver, we can see a signal for the TLB sets 25 and 27. These TLB sets have been selected based on their high slowdown for the last cacheline relative to all other cache lines. When testing for the four possible cache lines 3, 19, 35, 51 we can identify cache line 19 as the one containing the page table entries. Cache line 19 has been selected because it's not slow across all TLB sets but destintively in TLB set 25 and 27.

evictions, without detecting the jump in between cache lines and absent other side channels, it is challenging to detect if an eviction induced a cache miss during a page table walk or in other kernel operations.

### 7.3. Combining the two side channels

Next, we combine the two side channels used by TagBleed to 1) derandomize all the required bits of entropy and 2) combine the available information for better TLB set and cache line detection. We first showcase our TagBleed attack sensing the second page table level for a huge page access in user space. This scenario demonstrates our attack in a low-noise scenario due to the ability to carefully time a single user-level memory access. Next, we show that TagBleed's signal is still detectable when measuring it through kernel activity triggered by a system call.

Figure 8 depicts the signal for our combined EVICT+TIME attack on user-level huge page access. As shown in the figure, the attack yields a fast access for all TLB sets except the one triggering a page table walk. If, for that TLB set, we also evict the cache line of the second-level page table entry, the page table walk (and ultimately the access) is even slower. To only analyze the signal for the second-level page table, we keep the higher levels cached in the page table caches.

Figure 9 presents the same experiment but operated on an invalid syscall—a more useful but also more noisy scenario. As expected, the kernel activity yields several other memory access and results in several other cache misses. Nonetheless, we can identify the correct TLB set for kernel pages by finding the TLB sets that consistently keep getting evicted for all the cache lines. Another stratey to find the correct TLB set is finding several TLB sets close to each other. This evidences accesses within different parts of the kernel image in consecutive TLB sets. However, the most reliable way to determine the correct TLB set is verifying if one of the four possible cache lines is considerably slower than the other cache lines.

### 7.4. Success rate

In order to evaluate the success rate of TagBleed, we ran 50 trials, restarting the system each time to trigger a rerandomization with KASLR. For each trial, we performed a total of 20 runs to minimize the risk of temporary noise. In 47 of 50 trials, we are able to recover the correct location of the kernel. In three other trials, we could not reliably disambiguate two cache lines. This translates to a 94% success rate while reducing the KASLR entropy down to 1 bit in the other cases.

### 7.5. Attack time

Our TagBleed attack can be run in less than a second with satisfying results. Nonetheless, timing is usually not critical when building kernel exploits in a local exploitation scenario, where the attacker has already been granted (or achieved) unprivileged code execution. By default, we therefore increase the number of rounds to 10 for each TLB set to reduce the amount of false positives. This still allows us to run the attack in less 3 seconds including the time to run our solver script.

### 7.6. Noise

For the purpose of noise reduction, TagBleed combines two different side channels. Figure 7 shows that, without combining multiple side channels, TagBleed cannot easily battle spatial noise. For increased reliability against temporal noise, we repeat our measurements in several rounds. Temporal noise is especially critical when timing system calls. The kernel can potentially reschedule other processes after the system call instead of rescheduling the attacker-controlled process. By repeating the evictions in several rounds, we make sure that temporal noise does not negatively affect the measurements. As we lower

TABLE 4. KASLR ATTACKS VS. DEFENSES.

| | KAISER [5] | LAZARUS [6] | time |
|---|---|---|---|
| Double PF [2] | ✗ | ✗ | $<$ 1 min |
| prefetch [3] | ✗ | ✗ | 12 s |
| TSX [4] | ✗ | ✗ | 0.2 s |
| BTB [1] | ✓ | ✗ | 60 ms |
| TagBleed | ✓ | ✓ | $<$ 1s |

the number of rounds below the default value of 10, we quickly observed degradation of the success rate due to temporal noise. As we increase the number of rounds above 10, we observed minor improvements to the previously reported success rate. Overall, we believe 10 is a good choice even for relatively noisy environments.

To asses noise in a realistic use case, we ran the experiment with additional workload on a Ubuntu Desktop with an open browser, running a youtube video, and several applications running such as an email client. We could not measure any effect that suggests TagBleed is affected by such noise. The attack is also not required to run longer compared to the experiment without additional workload.

### 7.7. Comparison against other KASLR attacks

In Table 4, we compare our attack with existing attacks when considering existing state-of-the-art mitigations. TagBleed compromises KASLR regardless of either mitigation since it relies only on the shared tagged TLB and CPU data caches rather than shared branch state or shared address space. We verified our attack TagBleed against the latest Linux kernel with KPTI, which is based on the KAISER patches. All other existing attacks are mitigated by LAZARUS. Since TagBleed does not rely on a unified address space or the BTB, it is not mitigated by LAZARUS. Furthermore, KAISER protects against all existing attacks except BTB-based attacks (although the latter can be also mitigated using complementary mitigations such as eIBRS [12]).

## 8. Mitigations

We distinguish between defenses specific for the TLB side channels, defenses targetting cache side channels, and generic mitigations that target the timing primitives.

**Stopping TLB side-channel attacks.** Completely removing the TLB side channels requires all shared state to be removed. There are two ways to do so: spatial and temporal partitioning.

Spatial partitioning removes the side channel by isolating user processes from the TLB sets associated with the kernel. With current architectures this is not practically possible, since a partitioned TLB makes it extremely hard to guarantee contiguous virtual memory. Changing the TLB indexing function in future architectures could work, although hardware changes are expensive, and it is not unlikely that doing so will introduce performance degradation. After all, the current function is specifically chosen for performance.

Instead of spatial partitioning, it is also possible to kill the side channel by partitioning the TLB in time—by performing a full TLB flush upon crossing security boundaries. Disabling TLB tagging/PCIDs to flush the TLB completely effectively mitigates our attack but at the cost of high performance overhead for all implementations of kernel address space isolation.

**Stopping LLC side-channel attacks.** As temporal partitioning of the last level cache, although possible in theory, is not a feasible solution for obvious performance reasons, we consider only spatial isolation.

Cache coloring divides pages between kernel and user process in such a way that they do not share cache sets. Doing so will stop leaking through the LLC, but is far from trivial [34], has serious performance implications for both user processes and the kernel [35], [36], and needs to account for all memory used on behalf of the user process. Moreover, LLCs are not the only side channel option for the determined attacker.

Instead of caching the content of the page table in the LLC, future processor generations could put it in a dedicated, isolated page table cache. Clearly, such a solution requires expensive hardware changes. Also, that page table cache would have to be designed carefully, lest it opens up a new potential side channel (e.g., if it is shared between user processes and kernel).

**Stopping general kernel timing attacks.** At heart, our attack is possible because attackers can measure subtle timing differences in system call execution. Removing this ability would also stop the attack. To do so, we identify three approaches: detection of timing attacks, constant time system calls, and timer crippling.

HexPads [37] has shown that, in principle, performance counters can be used to detect ongoing side-channel attacks. However, it is hard to guarantee full mitigation and detection also introduces the risk of false positives and false negatives. While at first sight it may also seem possible to introduce a defense in the kernel to detect a high rate of failed system calls, say, such a solution would be naive. First, it is hard to be sure if the failed calls are really due to a side-channel attack. Second, attackers can easily make their attack more silent by extending its time and running it from separate processes. In order to mitigate the ability to time invalid system call the kernel could enforce a constant execution time for invalid system calls. Doing so would not influence performance during normal use, since invalid system call number are not typically used by normal applications. However this is also not very effective because we could just find a short valid system call as an alternative to an invalid system call. The alternative, making all system calls constant time is not a practical solution. The most obvious mitigation is to cripple the timers, for instance by removing the availability of high resolution timers such as `rdtsc`. Again, this solution is not realistic since high resolution timers are vital to many applications. Moreover, crafting *ad-hoc* high-accuracy timers, e.g.. using concurrent threads is always possible.

In summary, mitigating side channels for every single memory access is challenging and/or expensive. The simplest and most practical mitigation may be to simply use the higher bits of the virtual address for the operating

system's implementation of KASLR. Since our attack fully derandomizes PTL2, extending the randomized bits into PTL3 could work as a possible mitigation, even though moving the location of the Linux kernel may (and probably will) introduce unforeseen performance issues. We point out that with our technique we would still be able to derandomize PTL2 which removes those bits from the actual entropy of both KASLR and user-space ASLR.

## 9. Related Work

**Derandomizing (K)ASLR.** Derandomizing ASLR has been an active research topic as a fundamental primitive for code reuse attacks [38]. The simplest way to break ASLR is to leak code or data pointers with memory disclosure vulnerabilities [39], [40]. However, side-channel attacks showed that even without disclosures it is possible to derandomize the address space layout. These side-channel attacks use techniques such as control flow timing [41], [42], memory deduplication [43], [44], or CPU caches [32], [3], [2], [4], [1].

Hund et al. [2] showed three different scenarios for breaking KASLR by perfoming timing attacks on CPU caches and the TLB. Yang et al. [4] used Intel TSX to suppress exceptions that normally happen on faulty memory accesses. Since no page fault is raised but the transaction aborts and returns directly back to the user, the difference between invalid permissions or a missing mapping is measurable. Grus et al. [3] showed that the excution time of the prefetch instructions can be used to detect the existence of virtual mappings in the kernel region. Evtyushkin et al. [1] demonstrated that the BTB (Branch target buffer) leaks bits to break current KASLR implementations in Linux. Finally, in 2018 Meltdown [8] used a speculative execution vulnerability in Intel CPUs to read the entire virtual and physical address space. This of course also breaks both ASLR and KASLR.

Most of the presented KASLR attacks suggested better isolation between kernel and user space as a defense. The attacks are possible because the kernel is mapped into each user process address space. Gruss et al. [5] presented KAISER as a kernel page table isolation (now implemented as KPTI in Linux) with low performance impact. With KPTI the whole kernel is no longer mapped into each user address space which defends successfully against the presented attacks except Evtyushkin et al.'s BTB attack [45], which instead is mitigated by explicitly (as done by LAZARUS [6]) or implicitly (as done by eIBRS [12]) flushing the BTB on privilege switches. eIBRS [12] can additionally prevent cross-thread BTB attacks if SMT is enabled.

**Hardware timing side channels.** Physical shared resource may easily give rise to side channels. For instance, as early as 1996 when Kocher [46] presented his timing side channel on crypto primitives, Kelsey et al. [47] mentioned the idea of using the cache hit ratio as a side channel on large S-box ciphers to break cryptographic keys. This theoretical idea was formalized by Page [48] in 2002. Just one year later the first successful cache-based attack against DES was presented by Tsunoo et al. [49].

**EVICT+TIME.** [22], [23] attacks can only observe a single cache set per measurement and have been used to recover AES keys from the victim's process. Concurrently, two other papers presented cache attacks to leak cryptographic keys. Bernstein used a similar method to EVICT+TIME to break AES, which required reference measurements for a known key in an identical configuration to the victims [27]. The second paper, by Percival [28] presented a cache-based attack on RSA with SMT.

Meanwhile, the PRIME+PROBE [22], [24] and FLUSH+RELOAD [25], [26] attacks are able to observe the state of the whole cache which makes them popular due to its faster bandwidth compared to EVICT+TIME. PRIME+PROBE was utilized by several researchers to leak private keys [50], [51], keystrokes [52] and to read information from other processes or VMs on the same machine from JavaScript [53]. The FLUSH+RELOAD attack requires page sharing with the victim, for example by some sort of memory deduplication. However, it is more fine grained than PRIME+PROBE by measuring the exact cache line. Memory deduplication was deployed in most major operating systems which resulted in several FLUSH+RELOAD based attacks exploiting shared memory [54], [55], [56], [57]. Several others used the CPU cache to extract private keys from AES implementations [29], [30], [31]. All previous attacks focused on reading secret information across boundaries, either from other processes or other VMs running on the same physical machine. Gras et al. [32] broke ASLR within the JavaScript sandbox with an EVICT+TIME technique.

However not only CPU caches are an attractive target for side channels. TLBleed [15] showed that the TLB can also be used to leak sensitive information with all CPU cache defenses deployed. The BTB (Branch Target Buffer) has also been shown to leak sensitive information through side-channel attacks [1]. Pessl et al. [58] presented DRAMA a cross-CPU side-channel attack exploiting the DRAM row buffer.

**SGX enclave attacks with TLB as an attack vector.** Previous work has shown that the TLB is shared between the SGX enclave and the process it is running in [59]. Since they run in the same address space this means classic TLB attacks apply on SGX environment [15], similarly to breaking KASLR without KPTI enabled. Therefore tagged TLBs do not enable new leakage in such a case, but can be interesting if the address space is isloated like with AMD SEV [60].

**Defenses against timing side channels on CPU caches.** As CPU caches became a target for side-channel attacks, several defenses were proposed. All defenses focus on scenarios where several untrusted entities share hardware (e.g., multiple tenants in the cloud). Cache isolation is intended to protect a tenant against an attacker running on the same physical machine [61], [35], [36], [62], [63], [64], [65], [66]. Some of the existing defenses focus on isolating critical code sections and disallow leaking information through the cache while executing in isolation [35], [62], [66]. Others protect areas in memory from leaking information to the cache [64], [36]. Some claim to provide full isolation between untrusted VMs running on a multi-tenant system [61], [63], [65]. None of these defenses isolate the kernel against the user or fully defeat CPU cache side channels. Especially traces left in the cache by the MMU, not by the user itself, will not be affected by

any of these defenses. We rely on information in the cache that is cached on every single address translation by the MMU. Providing cache isolation is limited by resources and therefore can only be provided for a limited time to execute security sensitive sections. Address translations are always present, therefore not just a small portion of the code can be isolated.

**Concurrent work.** In concurrent work, Data Bounce [67] and EchoLoad [68] present side-channel attacks to bypass KASLR in face of KPTI (and absent CPU bugs like RIDL [11]). In contrast to TagBleed, both attacks rely on the current KPTI implementation leaving a few kernel pages mapped in the user-visible address space with the same KASLR entropy used for all the other kernel pages. As such, similar to traditional address probing attacks against KASLR [3], [2], [4], such attacks can probe for user-mapped kernel pages and indirectly infer that of the other kernel pages. In contrast, TagBleed's confused deputy attack can directly drop the entropy of kernel pages mapped and used only by the kernel, showing that even a perfect implementation of KPTI as well as the most recent mitigations against address probing attacks such as FLARE [68] are insufficient.

## 10. Conclusions

In this paper, we demonstrated that isolating the address space organization of the kernel from that of user processes is not enough to prevent attackers from breaking randomization in the kernel (KASLR). Ironically, the one feature that is commonly hailed as the performance saviour for kernel address space isolation, the presence of address space tags in modern TLBs, turns out to break its isolation guarantees. Our attack makes use of the fact that tagged TLBs allow attackers to observe kernel memory accesses if they occur in the same TLB set. Moreover, by reverse engineering the TLB architecture, we were able to infer part of the kernel's virtual address from knowing the TLB set. Complementing our side channel with second one (based on cache activity as a result of page table walks), we completely broke KASLR in the Linux kernel, even in the presence of advanced defenses and kernel page table isolation. In conclusion, since we demonstrated that we need to reconsider the current designs and counter measures are invariably expensive and typically complicated, we now know that the transition from a unified address space organisation to one where the kernel gets its own address space will be more costly than we thought.

## Acknowledgements

We would like to thank our anonymous reviewers for their feedback. We would also like to thank Ben Gras for his help with the project. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreements No. 786669 (ReAct) and No. 825377 (UNICORE), by Intel Corporation through the Side Channel Vulnerability ISRA, and by the Netherlands Organisation for Scientific Research through grants NWO 639.021.753, VENI "PantaRhei", and NWO 016.Veni.192.262. This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

## References

[1] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, (Piscataway, NJ, USA), pp. 40:1–40:13, IEEE Press, 2016.

[2] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 191–205, IEEE Computer Society, 2013.

[3] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing smap and kernel aslr," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 368–379, ACM, 2016.

[4] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with intel tsx," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 380–392, ACM, 2016.

[5] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings*, pp. 161–176, 2017.

[6] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A. Sadeghi, "LAZARUS: practical side-channel resilient kernel-space randomization," in *Research in Attacks, Intrusions, and Defenses - 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18-20, 2017, Proceedings*, pp. 238–258, 2017.

[7] G. Tene, "Pcid is now a critical performance/security feature on x86." https://groups.google.com/forum/m/#!topic/mechanical-sympathy/L9mHTbeQLNU, 2018.

[8] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pp. 973–990, 2018.

[9] 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 *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[10] 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 *Proceedings of the 27th USENIX Security Symposium*, USENIX Association, August 2018.

[11] 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 *S&P*, 2019.

[12] "Deep dive: Indirect branch restricted speculation." https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation.

[13] 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 *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 937–954, USENIX Association, 2018.

[14] N. Hardy, "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, pp. 36–38, Oct. 1988.

[15] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 955–972, USENIX Association, 2018.

[16] "Kpti/kaiser meltdown initial performance regressions." http://www.brendangregg.com/blog/2018-02-09/kpti-kaiser-meltdown-performance.html, Feb. 2018.

[17] "The current state of kernel page-table isolation." https://lwn.net/Articles/741878/, 2017.

[18] "Kva shadow: Mitigating meltdown on windows." https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows, 2018.

[19] "Intel 64 and ia-32 architectures software developer's manual, volume 3a: System programming guide, part 1 (table 4-13)." Order Number: 253668-060US, 2016.

[20] "Github linux source code: kaslr.c." https://github.com/torvalds/linux/blob/12ad143e1b803e541e48b8ba40f550250259ecdd/arch/x86/boot/compressed/kaslr.c#L836, 2019.

[21] "Comparing aslr between mainline linux, grsecurity and linux-hardened." https://gist.github.com/thestinger/b43b460cfccfade51b5a2220a0550c35#file-linux-vanilla, 2018.

[22] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, (Berlin, Heidelberg), pp. 1–20, Springer-Verlag, 2006.

[23] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, pp. 37–71, Jan. 2010.

[24] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, (Washington, DC, USA), pp. 605–622, IEEE Computer Society, 2015.

[25] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games – bringing access-based cache attacks on aes to practice," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, (Washington, DC, USA), pp. 490–505, IEEE Computer Society, 2011.

[26] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 719–732, USENIX Association, 2014.

[27] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[28] C. Percival, "Cache missing for fun and profit," in *Proc. of BSDCan 2005*, 2005.

[29] J. Bonneau and I. Mironov, "Cache-collision timing attacks against aes," in *Proceedings of the 8th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'06, (Berlin, Heidelberg), pp. 201–215, Springer-Verlag, 2006.

[30] O. Acıiçmez and c. K. Koç, "Trace-driven cache attacks on aes (short paper)," in *Proceedings of the 8th International Conference on Information and Communications Security*, ICICS'06, (Berlin, Heidelberg), pp. 112–121, Springer-Verlag, 2006.

[31] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned aes t-tables," in *Proceedings of the 4th International Conference on Constructive Side-Channel Analysis and Secure Design*, COSADE'13, (Berlin, Heidelberg), pp. 200–214, Springer-Verlag, 2013.

[32] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, Feb. 2017.

[33] "Microsoft: A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003," Sept. 2006.

[34] "Lkml: Page colouring." https://goo.gl/7o101i, 2003.

[35] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring.," in *DSN Workshops*, pp. 194–199, IEEE, 2011.

[36] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, (Bellevue, WA), pp. 189–204, USENIX, 2012.

[37] M. Payer, "Hexpads: A platform to detect "stealth" attacks," in *Proceedings of the 8th International Symposium on Engineering Secure Software and Systems - Volume 9639*, ESSoS 2016, (Berlin, Heidelberg), pp. 138–154, Springer-Verlag, 2016.

[38] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, pp. 2:1–2:34, Mar. 2012.

[39] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, (Washington, DC, USA), pp. 574–588, IEEE Computer Society, 2013.

[40] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *NDSS*, 2015.

[41] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point(er): On the effectiveness of code pointer integrity," in *2015 IEEE Symposium on Security and Privacy (SP)*, vol. 00, pp. 781–796, May 2015.

[42] J. Seibert, H. Okhravi, and E. Söderström, "Information leaks without memory disclosures: Remote side channel attacks on diversified code," in *In ACM Conference on Computer and Communications Security (CCS)*, 2014.

[43] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup est machina: Memory deduplication as an advanced exploitation vector," in *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, pp. 987–1004, Institute of Electrical and Electronics Engineers, Inc., 8 2016.

[44] A. Barresi, K. Razavi, M. Payer, and T. R. Gross, "CAIN: Silently breaking ASLR in the cloud," in *WOOT*, 2015.

[45] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pp. 693–707, 2018.

[46] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, (London, UK, UK), pp. 104–113, Springer-Verlag, 1996.

[47] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Comput. Secur.*, vol. 8, pp. 141–158, Aug. 2000.

[48] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," 2002.

[49] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of des implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 62–76, Springer, 2003.

[50] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, (New York, NY, USA), pp. 305–316, ACM, 2012.

[51] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$a: A shared cache attack that works across cores and defies vm sandboxing – and its application to aes," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, (Washington, DC, USA), pp. 591–604, IEEE Computer Society, 2015.

[52] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 199–212, ACM, 2009.

[53] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 1406–1418, ACM, 2015.

[54] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," pp. 299–319, 09 2014.

[55] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, (New York, NY, USA), pp. 990–1003, ACM, 2014.

[56] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, (New York, NY, USA), pp. 85–96, ACM, 2015.

[57] T. Allan, B. B. Brumley, K. Falkner, J. van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, USA), pp. 422–435, ACM, 2016.

[58] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-cpu attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 565–581, USENIX Association, 2016.

[59] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, (New York, NY, USA), p. 2421–2434, Association for Computing Machinery, 2017.

[60] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," 2016.

[61] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, (New York, NY, USA), pp. 77–84, ACM, 2009.

[62] B. A. Braun, S. Jana, and D. Boneh, "Robust and efficient elimination of cache and timing side channels," *CoRR*, vol. abs/1506.00189, 2015.

[63] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 871–882, ACM, 2016.

[64] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 406–418, March 2016.

[65] R. Sprabery, K. Evchenko, A. Raj, R. B. Bobba, S. Mohan, and R. H. Campbell, "A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds," *CoRR*, vol. abs/1708.09538, 2017.

[66] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, (Berkeley, CA, USA), pp. 217–233, USENIX Association, 2017.

[67] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, *et al.*, "Fallout: Leaking data on Meltdown-resistant CPUs," in *CCS*, 2019.

[68] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break it, fix it, repeat," 2020.