Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction

Andrei LUȚAȘ (vlutas@bitdefender.com), Dan LUȚAȘ (dlutas@bitdefender.com)

Abstract

Speculative-execution based attacks and side-channels are more and more common as disclosures continue to increase scrutiny by researchers in this field. In this whitepaper, we demonstrate a new type of side-channel attack based on speculative execution of the SWAPGS instruction inside the OS kernel. This attack is capable of circumventing all existing protective measures, such as CPU microcode patches or kernel address space isolation (KVA shadowing/KPTI). We practically demonstrate this by showing how the speculative execution of the SWAPGS instruction may allow an attacker to leak portions of the kernel memory, by employing a variant of Spectre V1.

Disclaimer

We assume the reader has knowledge of CPU internals (branch prediction, out-of-order execution, speculative execution, pipeline, and caches), OS internals (system calls, interrupt and exception handling and KPTI), and side-channels and speculative-execution attacks in-general. It is best to read about Meltdown, Spectre, L1TF and MDS before approaching this whitepaper. That material provides fundamental building blocks about such attacks that are used in this whitepaper, but not described in detail. Rather, in this whitepaper we focus on only how the newly discovered attack works.

Introduction

While side-channel attacks have been well-known for some time, speculative-execution based attacks are new, and signs indicate they will persist for some time. Some of the most famous ones to date are **Meltdown** [1], **Spectre** [2], **L1TF** [3] and **MDS** [4] [5] [6]. These vulnerabilities allow an attacker to break the basic memory isolation provided by the hardware to access data which would normally not be accessible. At the most fundamental level, these vulnerabilities rely on a feature common in modern CPUs called *speculative execution.* This feature allows the CPU to execute instructions before knowing whether their execution is required. For example, *branch prediction* can lead to speculative execution. Each time the CPU encounters a conditional branch (an instruction which redirects the execution to another address if a certain condition is met), the CPU has to decide whether that branch should be taken or not very early in the front-end of the CPU, before actually executing the branch (in order to know where to start fetching the next instruction). To enhance performance, the CPU attempts to *predict* the outcome of the branch, and it starts executing instructions from the address indicated by the branch condition. However, later on, during the actual execution of the branch instruction, the CPU determines if the predicted outcome is correct or not. If it is, the instructions fetched and executed speculatively are committed normally, as they fall within the normal execution path. However, if the CPU determines that it mispredicted the branch, it discards all the instructions that were fetched and executed until that point, and it resteers the front-end to the correct address. This is not normally an issue as the discarded instruction don't produce any architecturally visible results (register or memory modifications) until the CPU determines they are needed. However, they do produce microarchitectural changes which can be observed by an attacker – in particular, they can leave cache traces. These cache traces are enough to leak secrets from arbitrary security boundaries (for example, from one process to another, from the kernel memory to the user memory, from SGX enclaves or from VMX root to VMX non-root), forming the basic block of speculative execution attacks.

Mitigations for this class of vulnerabilities are tricky to implement, and they generally fall in 3 categories:

1. Hardware fixes, which are available only in newer CPUs which address the flaws directly in silicon;
2. Software mitigations, which are implementations made entirely in software; the best example here is Kernel Page Table Isolation (KPTI) [7], which isolates the kernel memory into a different virtual address space, thus rendering several speculative side-channel attacks, such as Meltdown, ineffective;
3. Microcode mitigations, which require hardware and software cooperation: the hardware vendor supplies a microcode patch to expose some new functionalities (for example, the Spectre v2, L1TF or MDS mitigations) which are then used by hypervisor or operating system vendors to mitigate the vulnerabilities.

Currently, all the side-channels noted above are mitigated by at least one of the three listed categories. However, in this whitepaper we present a novel side-channel attack which bypasses all known mitigations by abusing a poorly-documented behavior of a system instruction called **SWAPGS**. The newly discovered side-channel allows an attacker to leak some portions of the kernel memory space, which would normally be protected by KPTI.

The SWAPGS instruction

SWAPGS is a system instruction (which means it can be executed only in kernel mode), available in 64-bit mode, and is intended to be used by only the operating system to switch between two Model Specific Registers (MSRs): the **IA32_GS_BASE** (for simplicity, it will sometimes be referred to as **GS base**) and the **IA32_KERNEL_GS_BASE**. The Intel Software Developer Manual [8] describes the behavior of this instruction in detail (Volume 2: Instruction Set Reference, A-Z, CHAPTER 4 INSTRUCTION SET REFERENCE, M-U). This allows the kernel to quickly gain access to internal, per-CPU data structures, as soon as a transition is made from user-mode to kernel mode. The normal usage scenario is that during a user-mode process execution, **IA32_GS_BASE** points to the user-mode per-CPU data structure - the Thread Information Block (TIB) on Windows - while the **IA32_KERNEL_GS_BASE** points to the kernel per-CPU data structure, the Kernel Processor Control Region (KPCR). When an event which switches from user-mode to kernel-mode occurs (for example, a **SYSCALL**, an interrupt, or an exception), one of the first things the kernel does is execute the **SWAPGS** instruction to switch the two MSRs; make **IA32_GS_BASE** point to the kernel per-CPU data, and **IA32_KERNEL_GS_BASE** to point to the user-mode per-CPU data. When the **GS** segment register is used to access memory, the CPU automatically uses the value in **IA32_GS_BASE** as the base address. **IA32_KERNEL_GS_BASE** is never used directly in addressing, and can only be accessed via the **RDMSR**, **WRMSR** or **SWAPGS** instructions. In contrast, the current value in **IA32_GS_BASE** can also be modified from user-space by using the **WRGSBASE** instruction.

The **SWAPGS** instruction is used mainly at the entry point of a **SYSCALL** or interrupt handler. The following example is the **SYSCALL** handler on a 64 bit Windows:

```
nt!KiSystemCall64Shadow:
0f01f8               swapgs
65488924251070000    mov   qword ptr gs:[7010h],rsp
65488b242500700000   mov   rsp,qword ptr gs:[7000h]
650fba24251870000001 bt    dword ptr gs:[7018h],1
7203                 jb    nt!KiSystemCall64Shadow+0x24
0f22dc               mov   cr3,rsp
```

As one can see, in this case **SWAPGS** is the first instruction executed after the user-kernel transition. This is important since immediately after, the user-mode stack pointer **RSP** is saved in the structure pointed by the freshly loaded **GS**. Next, a check is made to see if KPTI is enabled for this particular process. If it is, the kernel-mode **CR3** will be loaded (it was previously loaded into **RSP** from the KPCR).

In contrast, here is the code which handles a page-fault exception:

```
nt!KiPageFaultShadow:
f644241001           test    byte ptr [rsp+10h],1
7462                 je      fault_from_kernel
0f01f8               swapgs
650fba24251870000001 bt      dword ptr gs:[7018h],1
720c                 jb      nt!KiPageFaultShadow+0x22
65488b242500700000   mov     rsp,qword ptr gs:[7000h]
0f22dc               mov     cr3,rsp
fault_from_kernel:   ...
```

This code is slightly different. It first checks to see if the exception originated in kernel-mode (the first **TEST** instruction), and if it didn't, it executes **SWAPGS** to load the KPCR address into the **GS** base register

(since this means the exception originated in user-mode, and therefore, the **GS** base would point to the user-mode TIB). The rest of the code is similar to the **SYSCALL** handler.

Another interesting gadget is the following:

```
f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
7503               jne     skip_swapgs
0f01f8             swapgs
654c8b142588010000 mov     r10,qword ptr gs:[188h]
65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
488b8920020000     mov     rcx,qword ptr [rcx+220h]
488b8930080000     mov     rcx,qword ptr [rcx+830h]
6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

This code sequence is present in all exception handlers. Its role is unclear since a breakpoint established on this **SWAPGS** instruction revealed that it is never hit (from neither KPTI-enabled processes, nor processes with KPTI disabled), and therefore, this instruction probably very rarely executes.

The exploit

Since **SWAPGS** can be executed speculatively inside user-mode, an attacker can leak the address of the per-CPU data, normally available to only the kernel. This is a variant of Rogue System Register Load, and seems to work on only Intel CPUs. While this by itself may not mean too much, it allows an attacker to mount a KASLR bypass attack, thus subverting one of the most basic anti-exploit measures employed in the kernel. This is very intuitive, and is not the subject of this white-paper. Instead, we focus on leaking kernel memory.

There are 2 main scenarios that we have identified; **SWAPGS** not getting executed speculatively when it should, and **SWAPGS** getting speculatively executed when it should not. We discuss both scenarios in the following sections. However, the second scenario is much more serious, and our exploit (and the rest of the whitepaper) is based on it.

Scenario 1: SWAPGS not getting speculatively executed when it should

In this scenario a mispredicted branch redirects execution to a **GS** based addressing but **without** executing the **SWAPGS** instruction, although it should have, immediately after a user-to-kernel transition. Let's consider again the following example:

```
nt!KiPageFaultShadow:
[1] f644241001           test    byte ptr [rsp+10h],1
[2] 7462                 je      fault_from_kernel
[3] 0f01f8               swapgs
[4] 650fba24251870000001 bt      dword ptr gs:[7018h],1
[5] 720c                 jb      nt!KiPageFaultShadow+0x22
[6] 65488b242500700000   mov     rsp,qword ptr gs:[7000h]
[7] 0f22dc               mov     cr3,rsp
fault_from_kernel:       ...
```

This code sequence is executed when a Page-Fault (#PF) takes place. First it tests the **CS** register saved on the stack to see if the #PF originated in kernel. If it did originate in kernel, it jumps over the **SWAPGS** instruction, as the **IA32_GS_BASE** already points to the KPCR (since we were already in kernel when the #PF took place, it is assumed **SWAPGS** had already executed). However, when the #PF originates in user-

mode, **SWAPGS** has to be executed to make **IA32_GS_BASE** point to the KPCR. If the branch is mispredicted, execution continues in kernel with the user-mode **IA32_GS_BASE** active instead, thus opening another door for exploitation. An analysis of the Windows kernel shows that this does not seem to be a problem in practice since immediately after the user-to-kernel transition, the user **CR3** is still active, and to gain access to the entire kernel memory space, the kernel **CR3** must be loaded. As one can see in the above code sequence, the kernel **CR3** is loaded only after the **SWAPGS** instructions, and it is a serializing instruction, meaning that it cannot execute speculatively (all the instructions before instruction 7 must be retired before executing it).

Scenario 2: SWAPGS getting speculatively executed when it shouldn't

More interesting things happen when the **SWAPGS** instruction is executed speculatively inside the kernel. This is particularly problematic if it is followed by **GS** based addressing, which speculatively accesses the user-mode **GS** instead of the kernel one (since **SWAPGS** speculatively switched **IA32_GS_BASE** with **IA32_KERNEL_GS_BASE**). We disassembled the Windows kernel to see what instructions usually follow the **SWAPGS** instruction, and were surprised. Let's consider again one of our previous examples, which is the base of our exploit, and will be referred to from now on as *the gadget* (note that each instruction is labeled, for easier explanations):

```
[1] f60596a1390001     test    byte ptr [nt!KiKvaShadow],1
[2] 7503               jne     skip_swapgs [4]
[3] 0f01f8             swapgs
[4] 654c8b142588010000 mov     r10,qword ptr gs:[188h]
[5] 65488b0c2588010000 mov     rcx,qword ptr gs:[188h]
[6] 488b8920020000     mov     rcx,qword ptr [rcx+220h]
[7] 488b8930080000     mov     rcx,qword ptr [rcx+830h]
[8] 6548890c2570020000 mov     qword ptr gs:[270h],rcx
```

As one can see, the **SWAPGS** instruction 3 is followed by multiple **GS** based addressing instructions. In addition, the **SWAPGS** instruction is preceded by a conditional jump 2, which may sometimes be mispredicted. This means that sometimes, the **GS** based instructions access user-mode memory, because they are executing speculatively after **SWAPGS**. If an attacker modifies in user-mode **IA32_GS_BASE**, the speculative region would basically access whatever address the attacker wrote in that register (for example, using the **WRGSBASE** instruction). For example, if an attacker writes the value **0x1000** in **IA32_GS_BASE**, the CPU will speculatively access address **0x1000+0x188=0x1188** twice. The most interesting part is that the value loaded from the attacker-controlled address is further dereferenced. In our example, whatever value is loaded from address **0x1188**, it will further be dereferenced by instruction 6, **mov rcx,qword [rcx+220h]**. If, for example, at address **0x1188** lies the value **0xCC000**, this **MOV** instruction will attempt to load a QWORD from address **0xCC000+0x220=0xCC220**. Furthermore, whatever value lies at address **0xCC220** will be dereferenced again by the next instruction 7, **mov rcx,qword [rcx+830h]**. The result is a triple attacker-controlled dereference. How can we use this to our advantage? We will see this in Variant 2. Leak arbitrary kernel addresses.

Variant 1. Test if a certain value is located at a given kernel address

The first method of abusing the speculative behavior of the **SWAPGS** instruction relies on the fact that the CPU speculatively does at least 2 memory accesses. These 2 accesses are sufficient to allow an attacker to test if a value is located at a selected address inside the kernel. This can be considered a *search primitive*, as it allows the attacker to search for certain values in kernel memory. However, not any random value can be searched, as we will soon see. Let's consider again the exploit gadget listed previously. If **SWAPGS** is executed speculatively, and the **GS** base ends up pointing to an attacker-controlled value - let's call it **K**

- then instructions 4 & 5 will load a QWORD from address **K+0x188** – let's call the loaded qword **Q**. Furthermore, instruction 6 will load a QWORD from the address **Q+0x220**. The main question is, what happens if the attacker previously allocated memory at address **Q+0x220**? The answer is pretty straightforward; if the attacker already mapped that address, then the CPU, in its attempt to access it, leaves a signal inside the data caches, as the contents of address **Q+0x200** would be cached. Now that the basic pieces of the puzzle are all present, we can construct the search primitive. Assuming the attacker wants to see if value **V** is located at kernel address **K**, he would do the following:

1. Allocate memory at address **V-0x220**
2. Write the kernel address **K-0x188** into the **IA32_GS_BASE** register, using, for example, **WRGSBASE**
3. Wait for an interrupt or generate a fault, which would transition from user to kernel
4. If branch 2 is mispredicted:
    a. **SWAPGS** will be speculatively executed
    b. Value **Q** will be loaded from address **(K-0x188)+0x188**
    c. **Q+0x220** will further be dereferenced
5. When control is passed back from kernel to user, the attacker would check to see if address **V** is cached. If it is indeed cached, this confirms that value **V** (with a cache line bias – the actual value will be between **V** aligned to a cache line, which is usually 64 bytes, and **V** aligned to the next cache line) is located at kernel address **K**. Otherwise, this could mean that either the value **V** is not present at kernel address **K**, or that simply the branch was not mispredicted, and speculative execution of the **SWAPGS** instruction was not triggered. For better accuracy, each address would be tested several times, until either a match is found, or there is a certain probability that the value is not there.

Variant 2. Leak arbitrary kernel addresses

The second variant of the **SWAPGS** vulnerability is more generic, in that it allows the attacker to infer the value located at a randomly selected kernel address. However, the restriction from the previous section still applies (i.e., values that don't fall within the addressable user-mode domain are not easily leakable). It is more challenging to leak arbitrary addresses since we don't know what addresses to map in user-mode in order to detect speculative kernel accesses. The naïve approach in this case is to employ a linear search algorithm, where obtaining the value at kernel address **K** would go something like this:

1. Spray the entire user-mode virtual-address space
2. Write the target kernel address **K-0x188** into the **GS** base register
3. Trigger or wait for a kernel transition
4. If the gadget gets executed speculatively, and the value **V** located at address **K** falls within the addressable user-mode domain, a cache signal will be left, as address **V** will be cached
5. When returning to user-mode, check which address **V** has been cached

Of course, this technique is only theoretical as it is impossible to fill the entire user-mode space with memory; the attacker would have to allocate 2^{47}=128TB of memory. Needless to say, this is not feasible in practice. A more realistic approach is to allocate large chunks of memory and iterate through the entire address space. Realistically, there are high chances that if the allocated chunks of memory exceed the size of the Last Level Cache (LLC), in an attempt to see which address within the chunk has been accessed speculatively, the signal would disappear, as it would be evicted from the cache by the repeated tests. However, using chunks of memory roughly equal to the LLC is good enough, as this is usually more than

4MB in size, going up to 32MB in high-end systems. This means that the entire user-mode address space can be checked in 2^{47}B/32MB=4M iterations, which starts to be feasible.

There, are however, even better approaches. Let's take a look at the gadget once again. Instruction 5 loads value **Q** from the kernel address **K**. Instruction 6 then dereferences **Q+220h** and loads a new value, **P**, which is further dereferenced by instruction 7 which loads the value located at **P+830h**. This gives us a great opportunity; instead of allocating, for example, a 32MB chunk of memory and iterating through all of it to see if any address within this chunk was cached, we could spray the entire 32MB chunk of memory with the address of a single test variable. If the gadget gets to execute speculatively, and if the value **V** at kernel address **K** is within that 32MB memory region, then instruction 7 would speculatively access our test variable. Instead of iterating the entire 32MB memory region to see if any address is cached, we can directly test our variable; if it is cached, then the value **V** is within the tested range. Otherwise, we can move on with the search. This method significantly speeds up the process. The final algorithm may look something like this:

1. Allocate a chunk of memory, **M**, with size **S** (where **S** will usually be selected depending on the LLC size)
2. Fill chunk **M** with the address of our test variable, **T-0x830** (note that we don't know beforehand the offset where we should spray **T**, as it can between 0 and 7; this can be overcome by trying each possible offset)
3. Flush the variable **T** from the cache, using the **CLFLUSH** instruction
4. Write the kernel address **K-0x188** to be leaked in **GS** base register using **WRGSBASE**
5. Trigger the gadget to be executed speculatively
6. Inside the kernel space, if the gadget indeed executes speculatively:
    a. Instruction 3 (**SWAPGS)** makes the attacker controlled **GS** base active (which points to the target kernel address **K)**
    b. Instruction 5 loads value **Q** from address **K**
    c. Instruction 6 loads value **P** from address **Q+0x220**
    d. Instruction 7 loads value **R** from address **P+0x830**
    e. If **Q** is in the range of the allocated chunk **M**, then **P** will be equal to our test variable **T**, which will be dereferenced to load **R**; therefore, **T** will be cached if the gadget is executed speculatively and if the value at kernel address **K** is in the interval described by memory chunk **M**
7. When returning to user-mode, the attacker will test to see if variable **T** is cached
    a. If variable **T** is cached, then the value located in kernel at address **K** is in the interval described by memory chunk **M** (for example, if **M** was allocated at address 0 and has a size of 32MB, then the value at kernel address **K** is in the range [0, 33554432]), and the attacker may further zoom into that region by reducing the size **S** to be allocated (for example, **S** may be cut in half at each step)
    b. If variable **T** is not cached after a certain number of tries, the next interval can be checked: **M=M+S** and **goto** step 1

By employing this technique, the attacker can sequentially reduce the search interval until it reduces the number of possible values to an acceptable range.

Challenges

Speculatively executing the gadget

From this perspective, the exploit can be thought of as a new variant of Spectre v1, since it relies on speculatively executing unexpected instructions after a branch was mispredicted. Directly controlling the outcome of a conditional branch which is located in kernel is impossible, but we determined that we can control it, to some extent. The conditional branch in the vulnerable gadget is executed after testing whether the **KvaShadow** kernel variable has bit 0 set. If bit 0 inside **KvaShadow** is set, the jump will be taken, and the **SWAPGS** instruction is skipped, and therefore, not executed. To have a high probability of success, we must be able to indirectly manipulate the outcome of this branch, and fool the CPU into thinking the branch is not taken, and therefore, to speculatively execute the **SWAPGS** instruction. To do so, good knowledge about the organization of the branch prediction unit is required. Agner Fog has an excellent resource [9] which describes in detail how various CPU units work. Unfortunately, it appears that little is known about how branch predictors are organized and how they work on Intel Haswell and newer CPUs. To save time, we decided for to use a brute-force approach instead of carefully reverse-engineering these aspects. The building blocks required to influence the speculative execution of our gadget in kernel are:

Trick the CPU into thinking the branch instruction 2 is not taken: branch confusion**.** As already mentioned, there's no way to directly do so, and there is little information regarding the organization of branch predictors on Intel Haswell CPUs and newer [10]. We therefore made the assumption that regardless of how it's organized internally, the CPU must somehow use the branch instruction address in order to look it up inside the Branch Target Buffer (BTB). We expect that on Haswell and newer, the BTB is organized as any other cache, having various sets and ways (and indeed [10] hints at a 4-way set associative organization for Haswell, perhaps it is similar on recent CPUs as well) – this is better illustrated in Figure *1* Typical cache access scheme. Therefore, some bits of the branch address are used to index a particular set inside the cache (usually lower order bits), and other bits inside the branch address (usually higher order bits) are used as a tag (perhaps after applying a hash function on various portions of the address). Little to no information is known about actual BTB access, but we expect to be able to evict the target branch from the vulnerable gadget by allocating a long sequence of conditional branches which are situated at the same page offset as the target branch. For example, if the target branch is located at offset **0xCEE** inside the memory page, we would allocate a large memory area (for example, several KB or MB in size), and inside each page of this buffer, at offset **0xCEE,** we place an identical conditional branch, but which is never taken. This has two effects. First, the actual target branch will most likely be evicted from the BTB, and second, if there is a collision between the tag bits inside the BTB, we would directly cause the branch to not be taken. In reality, all we care about is evicting the branch from the BTB, because the CPU would normally employ static prediction on branches it sees for the first time. A very good write-up [10] hints that Intel Haswell CPUs always predict newly seen branches as not-taken, which is exactly what we need. Intel Ivy-Bridge seems to weakly predict ahead as not taken, which is again what we want. The Intel Optimization manual [11] states in Chapter 3, Section 4, subsection 1, paragraph 3, that the static predictor would predict backward taken, forward not taken, which confirms the described findings, and is favorable for our exploit.

Make sure the KvaShadow variable is not cached: cache thrashing. This is easier to do in practice, but other CPU cores executing in parallel may interfere by caching it back whenever it's accessed. To make

sure this variable is not cached, we employ a technique very similar to the one described in the previous section, but instead of flushing the BTB, the data caches are flushed. In doing so, we simply determine the page offset of the **KvaShadow** variable, and allocate a large chunk of memory (at least the last-level cache in size), and thereby access that offset in each of our memory chunk's pages. This ensures, with sufficient probability, that the variable is evicted from the caches, and the conditional branch is not only mispredicted, but it will have to wait until the variable is read from memory (which should take several hundred clock cycles). This gives us enough speculative execution time to employ the attack. The typical cache access mechanism is illustrated in Figure *1* Typical cache access scheme.

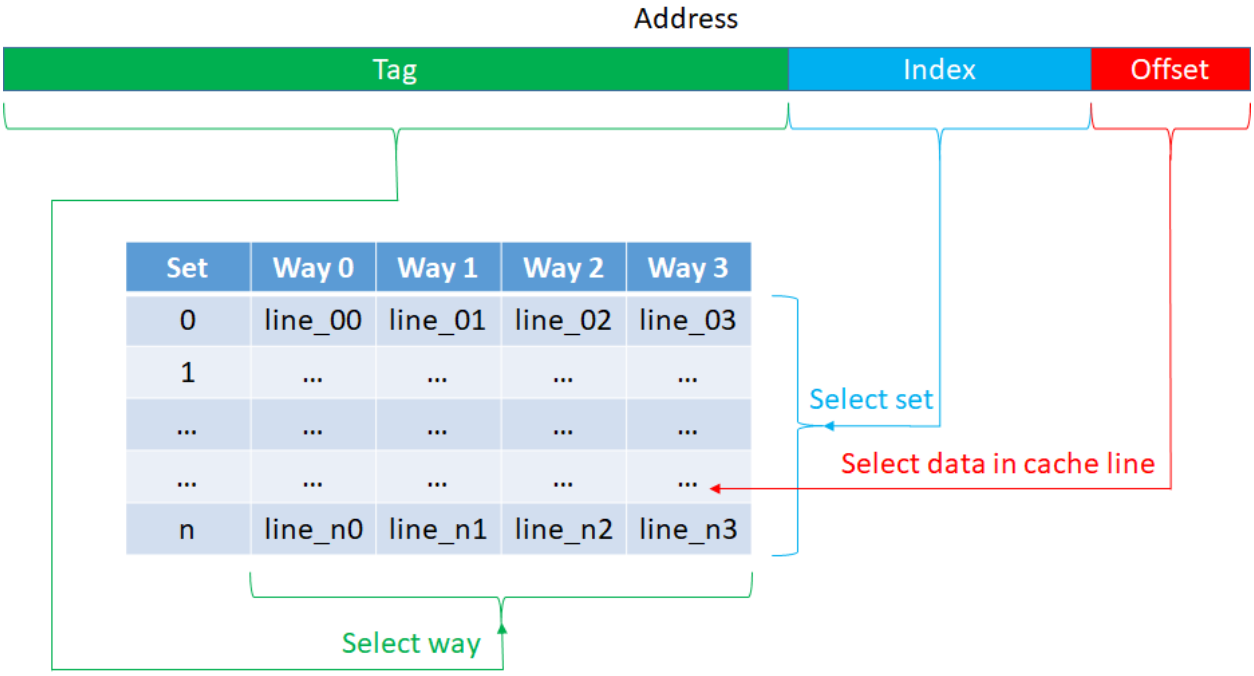

Figure 1 Typical cache access scheme

Test variable alignment

When we spray the decoy memory buffer with the address of the test variable (which, if cached, indicates that the kernel value lies within the tested memory interval), we don't know the actual alignment of the kernel value **Q** located at **K**. For example, if we spray address of the test variable **T** starting with offset 0 (**M + 0x310, M + 0x318, M + 0x320, M + 0x328**, etc.) inside the memory chunk **M**; but if the value **Q** at kernel address **K** is aligned to 3, for example **0x103**, the exploit will fail, as instruction 6 loads a value aligned to 3 (from **M + 0x323**, which does not contain the address of **T**). To overcome this, we can try each possible alignment since there are only 8 possible values (a QWORD is loaded from memory, which is 8 bytes in size). Modern CPUs have more than 2 cores, regularly 4 and even more than 8. The attack can be parallelized to run on each core, with a different alignment. Figure *2* A misaligned load would access other locations, instead of the test variable shows how a misaligned load would not produce the desired effect of accessing the test variable, and instead it would access another address.

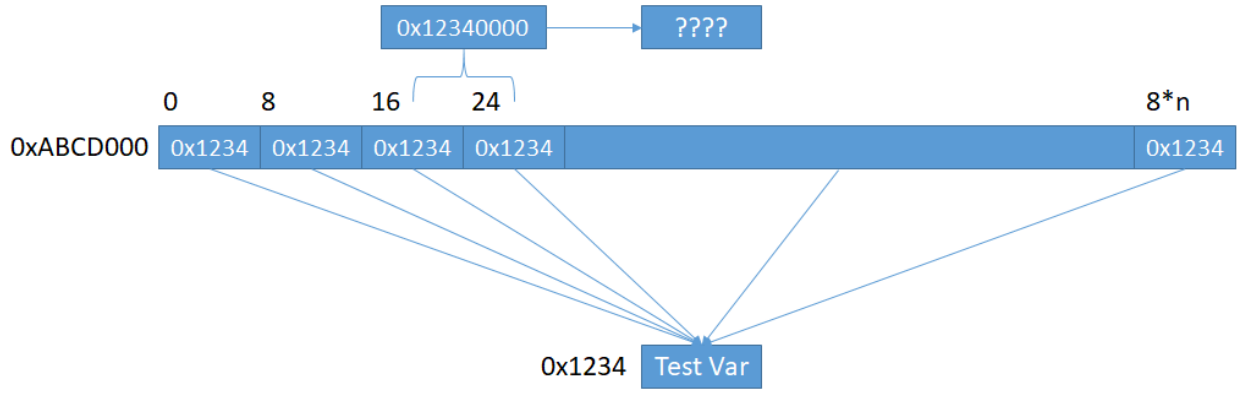

Figure 2 A misaligned load would access other locations, instead of the test variable

Cache Line Bias

Caches work with the granularity of a *line*, which is usually 64 bytes on modern CPUs. This means that if an address is cached, the entire 64 bytes region surrounding that address is cached. For example, if we wish to leak from a kernel address **K** which contains the value **0x123**, the region that is cached is **[0x100, 0x13F]**; therefore, we know that the value located at that address is in that interval, but we wouldn't know the exact value of the low order 6 bits. To identify the value of these low order bits, we can try to leak the value from kernel address **K-1** – this will translate into the value **0x123??**, where the question marks represent whatever byte value is located at address **K-1**. We call this technique *address shifting,* since we can shift 1 byte at a time from the kernel value which we dereference. The concept of cache line bias is illustrated in Figure *3* Cache line bias when inferring the kernel memory value.

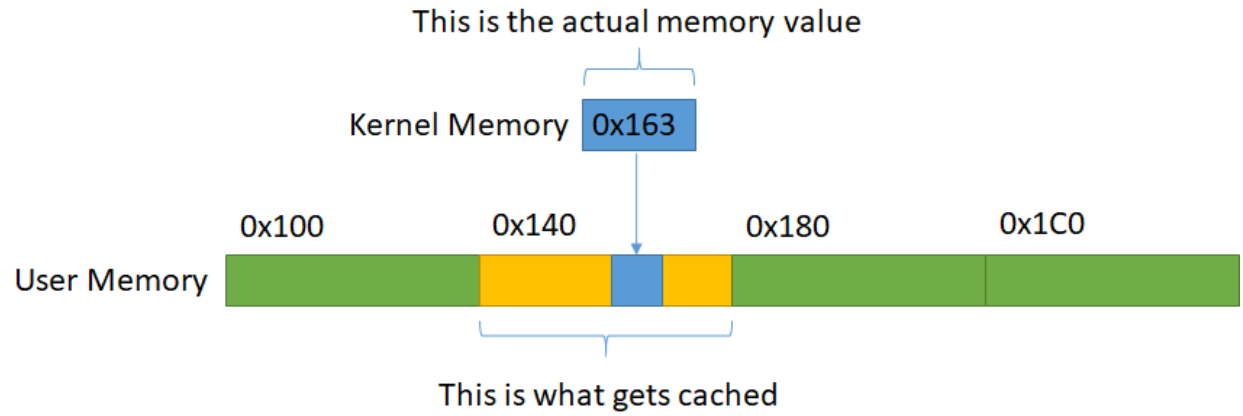

Figure 3 Cache line bias when inferring the kernel memory value

Leakable domain

The main disadvantage of this technique is that it cannot leak any arbitrary value; it can only leak values which resemble valid user-mode addresses. Due to the restrictions of 64-bit addressing, which uses 48-bit linear addresses, an address is considered to be a valid user mode address if it's in the range **[0x0000000000000000, 0x00007FFFFFFFFFFF]**. Intel announced recently that it wishes to extend the size of linear addresses from 48-bit to 57-bit, the **LA57** technology [12], which will extend this domain to **[0x0000000000000000, 0x007FFFFFFFFFFFFF]**. A quick statistic on a Windows RS5 ntoskrnl.exe memory

image revealed that **33.7%** of its contents can be leaked (approximatively 3.3MB out of 10MB). If we consider 57-bit addressing, about **51.2%** of the contents can be leaked (approximatively 5.1MB out of 10MB). We've also tested a random 10MB area of the non-paged pool, and we concluded that with 48-bit addressing, **55.9%** of the contents can be leaked, whereas with 57-bit addressing **65.8%** of the contents can be leaked.

Leaking other values

Although we didn't take the time to investigate further, we believe that any value which resembles a valid, canonical address may be leakable (although this should be more challenging). The rationale behind this is an article [13] which demonstrates how the ASLR can be bypassed by observing which cache sets are evicted by the Page Miss Handler (PMH) when performing a page-walk. In essence, the problem presented in this article – obtaining the value of an unknown virtual-address which is accessed by the attacker, is very similar to the problem we have; obtaining the value of random data accessed speculatively, and then dereferenced. This may be more problematic in practice, however, as in our case, there will be lots of noise, due to the necessity of transitioning in kernel each time to do an access of the secret value. We do believe that with enough time and resources, this method may be feasible in practice, and would allow an attacker to leak any canonical value. Given 48-bit linear addresses, this would increase the domain of leakable values to **37.7%,** and given 57-bit linear addresses, this would allow for **64.8%** of the nt image memory contents to be leaked. In practice, these two techniques may be combined. For the 10MB random non-paged pool area, this would allow for **65.0%** of the contents to be leaked with 48-bit addressing and **74.6%** with 57-bit addressing.

Performance

The performance of the attack varies greatly. Performance depends on how often the branch before the **SWAPGS** instruction is mispredicted, and the affected CPUs cache size. If an attacker is careful and patient enough, this may not be such a problem. However, in practice, we expect the variant 2 of the attack to be rather slow, with a speed not faster than a few bytes every few minutes, since it must search the value in a large space. Variant 1, however, is much faster. Confirmation for the presence of a given value at a tested kernel address takes place in well under 1 second. Since our goal was not to create a fully-functional exploit, but rather a PoC which proves the feasibility of leaking kernel values, we find the current performance acceptable. We anticipate that the leaking rate can be greatly improved by better controlling the mistraining of the branch prediction unit.

We measured the actual performance for Variant 1 (searching for a kernel value), with the following test:

Knowing that the first QWORD value inside the nt kernel image is **0x0000000300905A4D**, and knowing the base address of the nt image, we measured, on average, how much time and how many tries are required for the speculative gadget to be triggered and leave a measurable cache signal inside user-space. We mapped address **0x0000000300905A4D+0x220** in user mode, we wrote the kernel base in **IA32_GS_BASE** register using the **WRGSBASE** instruction, and we triggered a kernel transition by generating an Undefined-Opcode Exception (#UD) using the **UD2** instruction. Our measurement indicates that - on average - it takes about **0.0001 seconds**, or about **2 tries**, for the gadget to be triggered and leave the cache signal which confirms that a value within the interval [**0x0000000300905A40, 0x0000000300905A80**] is located at the tested kernel address. Of course, to maximize our chances of triggering speculative execution of the gadget, we employed the branch confusion and cache thrashing

techniques previously described. The test was conducted on a Windows 10 RS5 x64, powered by Intel Core i7-8650U with 8 logical cores.

Other operating systems and CPUs

The focus of our research was Microsoft Windows, as it was a low hanging fruit in terms of demonstrating the exploit. A quick analysis of the Linux kernel revealed that although it contains a gadget which may be used in an attack, it lies inside the Non-Maskable Interrupt (NMI) handler. We therefore believe that Linux would be difficult (if not impossible) to attack. A quick analysis of the Hyper-V kernel and of the Xen hypervisor kernel revealed that the **SWAPGS** instruction is not used, so exploitation is impossible. Other operating systems and hypervisors have not been investigated, although Microsoft, during the coordination of the disclosure, notified all the interested partied about this vulnerability. In addition, our PoC reliaes on the **WRGSBASE** instruction to modify the **GS** base in user-mode. This instruction is present starting with Ivy Bridge, and we expect that older CPUs to be much more difficult, if not impossible to exploit.

We tested two AMD CPUs: AMD64 Family 16 Model 2 Stepping 3 AuthenticAMD ~3211 Mhz and AMD64 Family 15 Model 6 Stepping 1 AuthenticAMD ~2100 Mhz and neither exhibited speculative behavior for the **SWAPGS** instruction.

Since the **SWAPGS** instruction is present only on x86-64, we don't expect other CPU architectures, such as ARM, MIPS, POWER, SPARC or RISC-V to be vulnerable. However, we don't exclude the existence of other similarly sensitive instructions that may execute speculatively.

Mitigations

The bad news is current mitigations such as microcode patches or KPTI do not address this newly discovered technique. The good news is there are several options for mitigating this vulnerability.

Clobber the user-mode GS on user-kernel transitions

One way to mitigate this vulnerability is by ensuring the user-mode **GS** base contains a known value, and not something controlled by the attacker. This must be done very soon after the transition, preferably before any conditional branches take place (which may allow exploitation still). However, this technique requires considerable work on the kernel side from all OS vendors. Considering there are simpler ways to mitigate this issue, it will probably never be leveraged as a fix.

Supervisory Mode Access Prevention

Supervisory Mode Access Prevention (SMAP) is a technology which prohibits user-mode pages from being accessed while in kernel mode. As the attack relies on speculatively accessing user-mode memory from kernel space to infer sensitive value, SMAP is more than capable of mitigating this issue. SMAP is already used by Linux kernels on CPUs which provide support. On Windows, SMAP requires significant engineering, since the driver model allows user-mode memory access by default.

Serialize the execution of the SWAPGS instruction

The most straight-forward way of mitigating this remains the serialization of the **SWAPGS** instruction. This can be done by placing an instruction such as **LFENCE** before or after each sensitive **SWAPGS** instructions. Normally, the modification itself is trivial, and the performance impact should likewise be minimal, as only the rarely taken branch is affected. However, care must be taken since this only covers scenario 2, where

SWAPGS is executed speculatively when it should not. To also cover scenario 1, a serializing instruction must be placed at the beginning of each block of code executed as a result of a branch skipping **SWAPGS**. This ensures that code is not executed speculatively without having executed **SWAPGS** beforehand, if it was required.

Hardware fixes

Of course, the most complete solution to this problem is to fix the CPU. Releasing a patch for the CPU is not as simple as releasing one for software. This will probably not happen very soon – perhaps future CPUs will be designed with avoiding this flaw in-mind such-that they disallow speculative execution of the **SWAPGS** instruction. Microcode updates are excluded as well, as Intel clearly stated when we initially reported the vulnerability that they do not wish to address this problem in affected CPUs.

Hypervisor based mitigations

Hypervisor Memory Introspection (HVI) is a technology that leverages CPU virtualization (Intel VT-x, for example) to provide new levels of protection. HVI analyses the memory of the guest virtual-machine (VM), identifies objects of interest, and uses technologies such as the Extended Page Table (EPT) to protect said objects against unauthorized access. Using the hypervisor, the vulnerable gadgets can be searched inside the OS memory, and they can be instrumented in order to make them safe (for example, by serializing them).

Conclusions

Speculative-execution based attacks are the new standard when it comes to cutting edge exploits and attacks. Fortunately, there aren't any widely known examples of these types of vulnerabilities being exploited in the wild. Perhaps this is because the community is highly mobilized to find and report these issues to vendors as soon as possible, or it's simply because they were not discovered yet. Overall, having this new category of attacks in the spotlight is beneficial, from a security standpoint, as many researchers focus on discovering new ways of abusing poorly understood behaviors or structures present in the CPU.

In this whitepaper, we presented a novel approach (a technique very similar to Spectre V1) of leaking sensitive information from the kernel. By abusing the fact that the **SWAPGS** instruction can be executed speculatively, one can force arbitrary memory dereferences in kernel, which leaves traces within the data caches. These signals can be picked up by the attacker to infer the value located at the given kernel address.

We have identified three main use cases for this technique:

1. Obtain the value of the **IA32_KERNEL_GS_BASE** from user-mode, and thus bypass KASLR
2. Search values in kernel memory – check if a given value is located at a given kernel address
3. Leak arbitrary memory – by employing a *divide et impera* technique, an attacker may be able to leak values from arbitrary kernel addresses

The advantage of this newly described technique is that it bypasses every known mitigation to date. The disadvantages are that it can leak only values which resemble valid user-mode addresses, and in the second use-case, it can be slow. However, since the introduction of **LA57** by Intel, the domain of *leakable* values increased from 47 bit to 56 bit. In addition, there have been attacks demonstrated which are capable of leaking portions of a virtual address by observing which sets have been evicted by the page-

walker when translating a linear address. Luckily, mitigations for this new technique can be implemented entirely in software, and they don't require microcode patches. Serializing **SWAPGS** execution mitigates this type of attacks. Furthermore, we used our Hypervisor Introspection solution to mitigate this vulnerability before patches were publicly available for it.

Glossary

- *pipeline* – technique used by modern CPUs, which involves splitting instruction execution into different stages (fetch, decode, rename, execute, write-back, etc.); modern CPUs have anywhere from 4 to 20 or 30 pipeline stages
- *out of order execution* – technique used by modern CPUs which allows them to execute instructions whenever the input data is available, rather than executing them in program order
- *speculative execution* – ability to execute instruction before knowing whether they are required or not
- *branch prediction* – technique used by modern CPUs in order to guess the outcome & destination of branches, so that instruction execution can continue before knowing whether the branch is actually taken or not
- *cache* – small & fast memory, placed very close to the CPU core, which contains data that was recently accessed (temporal locality) or data that is around recently accessed data (spatial locality); various types of caches may exist (data cache, instruction cache, micro-op cache) and levels (level 1, 2, 3, etc. – the higher the level, the bigger the cache capacity is and the slower the access time is)
- *instruction retirement* – when the CPU knows for sure the results of an instruction are valid (no fault was generated) and the instruction is not speculative, it will retire it, which means the results are written into the logical registers/caches/memory. Instruction retirement takes place in program order, which means the instructions *appear* to execute in the order in which they were written

Timeline of the discovery

07 August 2018 – Notified Intel that the **SWAPGS** instruction can be executed speculatively in user-mode, which allows an attacker to leak the address of sensitive kernel-mode structures, such as KPCR on Microsoft Windows

29 August 2018 – Intel responded that the behavior of the **SWAPGS** instruction is known, and that they do not intend to address it in affected CPUs

21 September 2018 – Insisting that this behavior is problematic, and it should be addressed

08 October 2018 – Intel responded that their position regarding a potential KASLR bypass remains unchanged

29 March 2019 – Reported to Intel that the speculative behavior of the SWAPGS instruction, if triggered in kernel mode, allows an attacker to bypass KPTI and thus leak kernel memory

01 April 2019 - Intel responded and said they've started investigating

02 April 2019 - Intel confirmed the issue but worked with ecosystem partners to mitigate at the OS kernel level. They connected us with Microsoft who agreed to coordinate with others in the industry to address the issue at the software level.

03 April 2019 – Reported to Intel that the Linux kernel contains vulnerable gadgets as well, though a PoC was not developed, and the complexity of an exploit is unknown

03 April 2019 – Intel responded that they will investigate, and that we should let them approach Linux kernel dev community

10 April 2019 – Got into contact with Microsoft, and was asked for more technical details

16 April 2019 – Provided Microsoft the requested technical details

17 April 2019 – Microsoft responded that they were investigating

18 April 2019 – Microsoft responded that they believe the gadget cannot be used to leak arbitrary memory

22 April 2019 – Provided Microsoft a new PoC, which demonstrated the ability of leaking arbitrary memory

23 April 2019 – Microsoft responded that they were investigating

30 April 2019 – We ask Microsoft if they have any updates

01 May 2019 – Microsoft responded that they have finished reviewing the report, and that they are waiting for OS team feedback, and are discussing with Intel regarding coordination

07 May 2019 – We ask Microsoft if they have any updates

07 May 2019 – Microsoft confirms that they reproduced the report and they are targeting a July patch, but this is subject to change, depending on how coordination goes

14 May 2019 – We ask Microsoft if they have any updates

15 May 2019 – Microsoft responded they are wrapping up the Microarchitectural Data Sampling issue from Intel, and that they will provide updates soon

24 May 2019 – Microsoft said they made good progress with the investigation, and that they are targeting mid-summer/late summer for the fix; also, they are talking with Intel regarding industry coordination

24 May 2019 – Notified Microsoft that we intend to present our findings at BlackHat

05 June 2019 – We ask Microsoft if they have any updates on the BlackHat presentation part

06 June 2019 – Microsoft responds they made good progress and are beginning the coordination with the interested industry vendors; they ask how much advance notice we need for BlackHat

06 June 2019 – We specify that we do not wish to present a 0-day, and we wish for all affected vendors to have time to address the issue before publishing anything

11 June 2019 – Microsoft asks if they can approach Linux and if we have a PoC for Linux

12 June 2019 – We respond we are okay with approaching Linux, and that we will see if a Linux PoC is doable

13 June 2019 – We notify that a Linux PoC is much more difficult to implement than a Windows one, and they should approach them without a PoC

18 June 2019 – Microsoft confirms they will approach Linux without a PoC

19 June 2019 – Microsoft asks for our explicit permission to reach Linux Kernel Devs and other vendors

19 June 2019 – We explicitly offer our permission

19 June 2019 – Microsoft asks if we agree with a general summary of the issue

19 June 2019 – We agree with the general summary

25 June 2019 – We ask for updates, specifically: if they notified the community, if they notified AMD, if they assigned a CVE number, and if they have a release date for the patches

28 June 2019 – We receive the update, stating that Linux is still working on mitigations, AMD was involved, there is a tentative CVE-2019-1125 (but not sure if Microsoft will issue it), and that the disclosure date is not final yet. We are asked if we still wish to present at BlackHat

28 June 2019 – We ask if AMD confirmed the issue, and we confirm we still with to present at BlackHat; we also state that we are open towards helping for a better community sync, if needed

28 June 2019 – Microsoft states that they may have a definitive answer from AMD by the beginning of July; they also state the tentative date for the fix – 9th of July. They also throw the idea of a possible delay for the fix, since not all parties may address the problem in time. However, they clearly state the intended date for the fix to be 9th of July

28 June 2019 – We state that our PR/Mrkt teams are pushing for the 6th of August, and that would be the worst case scenario date for us. We state that we are working on a technical white-paper describing the problem and how it can be abused

28 June 2019 – Microsoft stated that some parties may not be ready before 6th of August, and if it's acceptable for us to push the date

28 June 2019 – We state that we do not wish to jeopardize anyone, and we express our concern regarding the coordination process, as it takes too long, since the issue is already almost a year old

28 June 2019 – Microsoft asks for the timeline, as they did not know the issue is this old

28 June 2019 – We provide the timeline, starting with the initial reporting to Intel, in August 2018

28 June 2019 – Microsoft compliments Bitdefender on putting the safety first, even if this means wasting a great chance, such as presenting at BlackHat

28 June 2019 – We kindly ask how did Microsoft end up handling this kind of (hardware) issue

01 July 2019 – Microsoft exposes the reason behind them being in charge with this case. They also state they will release the patches on July 6th, but won't document the fix publicly until August 6th, to leave enough time for other vendor to test and deploy their fixes. They also ask what we plan to do in case of a tip-off

02 July 2019 – We responded that in the case of a tip-off, we should have a coordinated communication plan, and we ask what their plan is in this regard

10 July 2019 – We send this whitepaper for review

10 July 2019 – Microsoft acknowledges us sending the whitepaper, and requests permission to share it with the community

10 July 2019 – We confirm we agree with the whitepaper sharing

10 July 2019 – Microsoft thanks us, and indicates they will review it, and send feedback, if needed

19 July 2019 – Microsoft reports that everything is going as planned for the 6th of August releases, and that they would keep us posted, anything should change

20 July 2019 – We confirm our PR/comm teams are prepared for the 6th of August release; if anything should intervene, we must synchronize their and out PR/comm teams

30 July 2019 – We ask if they reviewed the whitepaper, and if there is any feedback

30 July 2019 – Feedback is provided, the whitepaper is adjusted accordingly

06 August 2019 - Public disclosure.

References

[1] M. a. S. M. a. G. D. a. P. T. a. H. W. a. F. A. a. H. J. a. M. S. a. K. P. a. G. D. a. Y. Y. a. H. M. Lipp, "Meltdown: Reading Kernel Memory from User Space," *Proceedings of the 27th USENIX Conference on Security Symposium, https://meltdownattack.com/meltdown.pdf,* 2018.

[2] D. G. D. G. W. H. M. H. M. L. S. M. T. P. M. S. Y. Y. Paul Kocher, "Spectre Attacks: Exploiting Speculative Execution," *CoRR, https://spectreattack.com/spectre.pdf,* 2018.

[3] J. V. B. M. M. D. G. B. K. F. P. M. S. R. S. T. F. W. a. Y. Y. Ofir Weisse, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient," *https://foreshadowattack.eu/foreshadow-NG.pdf,* 2018.

[4] D. M. M. L. M. S. J. V. B. D. G. D. G. F. P. B. S. Y. Y. Marina Minkin, "Fallout: Reading Kernel Writes From User Space," *https://mdsattacks.com/files/fallout.pdf,* 2019.

[5] M. L. D. M. J. V. B. J. S. T. P. D. G. Michael Schwarz, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," *https://zombieloadattack.com/zombieload.pdf,* 2019.

[6] A. M. S. Ö. P. F. G. M. K. R. H. B. C. G. Stephan van Schaik, "RIDL: Rogue In-Flight Data Load," *https://mdsattacks.com/files/ridl.pdf,* 2019.

[7] M. L. M. S. R. F. C. S. M. Daniel Gruss, "KASLR is Dead: Long Live KASLR," *https://gruss.cc/files/kaiser.pdf,* 2017.

[8] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual," *https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf,* 2019.

[9] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs," *https://www.agner.org/optimize/microarchitecture.pdf,* 2019.

[10] M. Godbolt, "Branch prediction," 2016. [Online]. Available: https://xania.org/201602/bpu-part-one.

[11] Intel Corporation, Intel® 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[12] Intel Corporation, "5-Level Paging and 5-Level EPT," *https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf,* 2017.

[13] K. R. E. B. H. B. C. G. Ben Gras, "ASLR on the Line: Practical Cache Attacks on the MMU," *https://www.cs.vu.nl/~giuffrida/papers/anc-ndss-2017.pdf,* 2017.

[14] C. C. a. J. V. B. a. M. S. a. M. L. a. B. v. B. a. P. O. a. F. P. a. D. E. a. D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," *CoRR, https://arxiv.org/pdf/1811.05441.pdf,* 2018.

[15] J. Horn, "Reading privileged memory with a side-channel," Google, January 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html.