A TSX-Based KASLR Break: Bypassing UMIP and Descriptor-Table Exiting

Mohammad Sina Karvandi[1], Saleh Khalaj Monfared[1], Mohammad Sina Kiarostami[2], Dara Rahmati[3], and Saeid Gorgin[1]

[1] School of Computer Science, Institute For Research in Fundamental Sciences (IPM), Tehran, Iran, {karvandi,monfared,gorgin}@ipm.ir
[2] Center for Ubiquitous Computing, Faculty of ITEE, University of Oulu, Oulu, Finland, mohammad.kiarostami@oulu.fi
[3] Computer Science and engineering Department, Shahid Beheshti University, Tehran, Iran d_rahmati@sbu.ac.ir

Abstract. In this paper, we introduce a reliable method based on Transactional Synchronization Extensions (TSX) side-channel leakage to break the KASLR and reveal the address of the Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT). We indicate that by detecting these addresses, one could execute instructions to sidestep Intel's User-Mode Instruction Prevention (UMIP) and the Hypervisor-based mitigation and, consequently, neutralized them. The introduced method is successfully performed after the most recent patches for *Meltdown* and *Spectre*. Moreover, we demonstrate that a combination of this method with a call-gate mechanism (available in modern processors) in a chain of events will eventually lead to a system compromise despite the limitations of a super-secure sandboxed environment in the presence of Windows's proprietary Virtualization Based Security (VBS). Finally, we suggest software-based mitigation to avoid these issues with an acceptable overhead cost.

Keywords: Cache Side-channel · TSX · Meltdown · KASLR.

1 Introduction

As signs of progress in computer science, from Artificial Intelligence [15] to High-Performance Computing [9, 26] continues, the role of computer security research in both hardware and software is drawing more attention to the research community. Recently discovered microarchitectural vulnerabilities in modern CPUs, are known to be devastating. They are very easily implemented, practical, and most likely independent from the operating system, which makes them an imminent threat to computer privacy. Among them, speculative-execution based and side-channel attacks are more ubiquitous as new disclosures continue to showcase the increasing failure of secured design in the computer hardware [2]. These attacks are capable of circumventing all existing protective measures, such as CPU microcode patches, kernel address space isolation (Kernel Virtual Address (KVA), shadowing, and Kernel Page-Table Isolation (KPTI)). While side-channel attacks have been well-known for a relatively long time, speculative-execution based attacks are contemporary, and pieces of evidence indicate that they will persist for some time in the future.

Pioneered by Meltdown [22] and Spectre [18] attacks, numerous variations, and extension of microarchitecture vulnerabilities have been found, and their corresponding exploitation has proposed latterly. ForeShadow [35], MDS [24], and ZombieLoad [29] should be alluded as the most famous ones. Moreover, new works have shown the extensiveness of these attacks. As an example, NetCAT [20] presents a practical network-based side-channel attack.

After Meltdown, more strict KASLRs such as KAISER [5] have been employed in today's operating systems to prevent similar attacks since short-term hardware mitigation is not effortlessly attainable. KAISER completely isolates the user-mode and kernel-mode memory layout by creating a *Shadow* representation of the mapped memory. However, there are still some unprotected addresses and parts by KALSR that required by the architecture. Hence, knowing these structure's addresses could lead to severe problems. In addition, discovered hardware-based vulnerabilities on Memory (DRAM) such as RowHammer [17] allow attackers to execute more destructive and offensive malicious code, to trespass or gain access to restricted and private information [32].

Furthermore, it is possible and suitable to take advantage of some hardware-specific structures that are implemented across operating systems. In the same way, once can gather masked and hidden internal information of the operating system which could be used for malicious purposes. To be more precise, the structures of Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT) are one of the essential parts of protected mode, which are not heavily isolated in the user-mode and kernel-mode address layout. By overwriting these structures in certain conditions, one can perform a privilege escalation attack. Also, by the use of the same variations of timing side-channel attacks as in Meltdown,(e.g., TSX-based attacks), the virtual addresses of these structures in the kernel memory could be revealed.

In this work, we demonstrate that GDT and IDT addresses could be discovered by TSX side-channel to perform privilege escalation attacks, even after Meltdown mitigation, bypassing the mitigations in modern Intel processors, particularly User-Mode Instruction Prevention (UMIP). Furthermore, it is illustrated that the proposed attacks can be executed in virtualized environments, such as the latest Microsoft Hypervisor release (Hyper-v) and Virtualization Based Security (VBS). In summary, the contributions of this paper are as follow:

- A concrete TSX side-channel attack is performed to discover *GDT* and *IDT* addresses in the kernel mode in a system with *KAISER* isolated memory layout bypassing UMIP.
- We show that a full system compromise could be achieved by revealing *GDT* and *IDT* virtual addresses in the memory, incorporated with *call-gate* mechanism along with a conventional *Write What Where*.
- The possible mitigation investigated for this vulnerability and low-cost software-based mitigation for the operating systems to avert these attacks is suggested.

2 Preliminaries and Background

In this section, required preliminaries and background knowledge for the software-based side-channel attacks, along with some concepts KAISER, TSX side channels, UIMP and Descriptor-Table Exiting are reviewed.

2.1 KASLR, Meltdown and KAISER

The security of computers highly relies on memory isolation, meaning that the kernel address ranges are not meant to be accessible from user prospective. The most conventional method to address such requirement is the Kernel Address Space Layout Randomization (KASLR) which include the random assignment of kernel objects rather than constant addressing. Discovered Meltdown [22] attack was able to exploit side effects of out-of-order execution on modern processors to read arbitrary kernel memory locations, including crucial personal information and passwords. By exploiting the out-of-order execution as an indispensable performance feature, the attack is independent of the operating system, and it does not rely on any software vulnerabilities. Meltdown was able to break all the security considerations provided by address space isolation as well as the virtualized isolation developed by the same infrastructure. The affected systems by Meltdown include a wide range of personal computers, smart phones, and even the enterprise cloud servers. Moreover, available TSX technology in Intel CPUs enables Meltdown to read the protected kernel memory addresses with the high-performance speed of 500Kbps. [22].

Generally, Meltdown mitigation relies on isolating kernel and user memory pages with different methods. The widely used approach to address this issue is the employment of KAISER [5], which is implemented as Kernel Virtual Address Shadow (KVAS) (a term coined by Microsoft) [23] in Microsoft Windows and KPTI in Linux [4]. In KAISER, placing a small portion of information in the user-mode is inevitable since operating systems are required to implement functions necessary to handle system calls and interrupts, which are directed to kernel space.

As will be discussed, leaving the tables which hold the addresses of interrupt handler (e.g., Interrupt Descriptor Table) or other tables managing the segmentation (e.g., GDT) visible to user mode, and ignoring to protect their addresses, allow the attacker to endanger the system. However, to adversely take advantage of the information left unprotected in the user-mode, essential internal mechanisms should be known which will be explored later.

2.2 TSX Cache Attack

Intel TSX refers to a product name for two x86 instruction set extensions, called Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM) [33]. HLE is a set of prefixes that could be added to specific instructions. These prefixes are backward-compatible. Hence, the code including them, also works on older hardware platforms. On the other hand, RTM is an extension adding

several instructions to the instruction set that are used to declare regions of code that should execute as part of a hardware transaction. A RTM transaction comprises the region of the code that is encapsulated between a pair of *xbegin* and *xend* instructions. Instruction xbegin also provides a mechanism to define a fall-back handler that is called if the transaction is aborted. *xabort* can be used by the executing code to abort the transaction explicitly. By employment of the TSX, generating an exception or an interrupt which is handled in the kernel could be avoided, resulting in side-channel attacks to be more resistant to noise with a more reliable outcomes [22]. As will be explored later on, We employ TSX to trigger the initialization of our proposed attack.

2.3 Descriptor-Table Exiting

Descriptor-Table Exiting is a hardware mechanism to restrict guest machines in VMX Non-Root from executing instructions such as LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, and STR [6]. This mechanism has been used in Microsoft Virtualization Based Security as an exploit mitigation, which avoids memory address leakage and provides an absurd situation for the attacker to find the base address of GDT or IDT, among other details such as Control Registers. Microsoft uses hypervisor as a hardware security mechanism, and in VM Control Structure. In order to configuring this hardware feature, an special field is presented which is referred as the *Descriptor-Table Exiting*. Descriptor-Table Exiting is declared in Intel Manual [6]. This control field determines whether executions of LGDT, LIDT, LLDT, LTR, SGDT, SIDT, SLDT, and STR cause VM exits. This declaration would be critical to the attack model we intend to describe.

2.4 User-Mode Instruction Prevention (UMIP)

UMIP is a security feature present in new Intel Processors. If enabled, it prevents the execution of particular instructions if the Current Privilege Level (CPL) is greater than 0. If these instructions were executed when CPL > 0, user space applications could have access to system-wide settings such as the global and local descriptor tables, the task register and the interrupt descriptor table. These are the instructions covered by UMIP in accordance to the Intel [6]:

- SGDT: Store Global Descriptor Table, SIDT: Store Interrupt Descriptor Table
- SLDT: Store Local Descriptor Table , SMSW: Store Machine Status Word
- STR: Store Task Register

If any of these instructions are executed with CPL > 0, a general protection exception (GP) is issued when UMIP is enabled. In order to enable this feature, operating systems can set the 11^{th} bit of the CR4. [6]

3 Attack Primitives: GDT and Call-Gate Mechanism

Our proposed attack is fundamentally based on the existing hardware features in the processors. As an indispensable part of the suggested attack, GDT and

its properties are described in detail in the this section. Moreover, here we discuss how the improper configuration might create vulnerabilities caused by the existing and possibly disabled hardware features.

3.1 Global Descriptor Table

GDT is an important data structure available in Intel x86-family CPUs providing the characteristics of the memory areas used during program execution. It includes the base address, the size, and access privileges which is fundamental in terms of the security prospective. GDT is a main table in x86 and protected-mode that still exists in AMD64 [1] and Intel IA-32e. The GDT structure in the x86 system is shown in Figure 1.

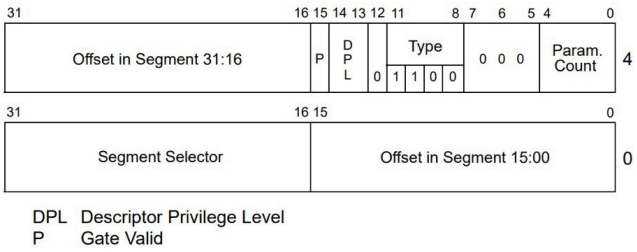

Fig. 1. GDT structure in a 32-bit machine

While the proposed attack here works on both x86 and x64 architectures, we have used the x64 version of GDT since it is more widespread rather than the other version.

3.2 GDT in 64-bit

Although the segmentation is omitted in the modern systems in protected-mode with paging enabled, the GDT still presents in 64-bit mode. A GDT must be defined but is generally never changed or used for segmentation. The size of the register has been extended from 48 to 80 bits, and 64-bit selectors are always *Flat* (thus, from `0000000000000000` to `FFFFFFFFFFFFFFFF`) which should also be taken into account when the attack chain is designed.

On the other hand, 64-bit versions of Microsoft Windows forbid hooking of the GDT. Attempting to do so would cause the machine to *bug check*. In our circumstances, it is not a problem for our case as long as some mechanisms for preventing these hooks called Kernel Patch Protection exists. This mechanism is known as *PatchGuard*, whihc checks the system in random intervals of between 3 to 10 minutes. So it is possible to patch GDT in a glance then revert everything to the privious normal state again to avoid such errors. In this context, we use GDT as a descriptor for call-gate to complete the attack chain instead of a descriptor for segmentation.

3.3 Call-gate Mechanism

Call-gates are used to transfer the execution to other rings e.g., ring 0, 1, 2, 3. Instructions like *SYSENTER* and *SYSCALL* are used in modern operating systems for transitioning between every rings to ring 0. But for the transition between other rings (e.g., ring 3 to 2 or 2 to 1), the call-gates would be used. The *type* field located in the GDT structure as indicated in Figure 1 represents a 4-bit field that could get various values and completely change the GDT entry behavior and definition. [13]

After finding the target entry, the type value should be changed to one *Gate* accordingly. For example, we use *0xc* (1100 - 32-bit call-gate) in the final payload. There are specific terms in call-gate used to build the final payload. In order to exploit the features that call-gate provides, the suitable privilege level should be set in the data segmentation used in the GDT. Here are the privilege levels defined in this context:

- **Current Privilege Level (CPL)** CPL is stored in the selector of currently executing the CS register. It represents the privilege level (PL) of the currently executing task and also PL in the descriptor of the code segment and designated as Task Privilege Level (TPL) [13].
- **Descriptor Privilege Level (DPL)** It is PL of the object which is being attempted to be accessed by the current task or put differently, the least privilege level for the caller to use this gate [13].
- **Requester Privilege Level (RPL)** It is the lowest two bits of any selector. It can be used to weaken the CPL if craved [13].
- **Effective Privilege Level (EPL)** It is maximum of CPL and RPL thus the task becomes less privileged [13].

Fundamentally, any task in an arbitrary code needs to fetch the data from the data segment. Therefore, the privilege levels are checked at the time a selector for the target segment is loaded into the data segment register. Three privilege levels are invoked into the privilege checking mechanism. Ultimately, the payload must meet the following conditions in the fields:

- RPL of the selector of the target segment.
- DPL of the descriptor of the target segment

Note that the access is allowed only if *DPL* is greater than or equal to the maximum of CPL and RPL, and a procedure can only access the data that is at the same or less privilege level.

3.4 From call-gate to code Execution in Ring-0

Call-gate in x86 In order to use x86, fields of a unique set of call-gate should be filled as described in Table 1.

Selector field should be *0x8* to point to KGDT_R0_CODE entry of GDT, which describes the kernel-mode in Windows. The type of it should be set to *0xc*, and the minimum ring that can invoke this call-gate is *0x3* (DPL = 0x3 (user-mode)), and also, it should be present in memory (*pFlag* = *0x1*) [13].

Table 1. Organization of the fields in the GDT

selector	0x8
type	0xc
dpl	0x3
pFlag	0x1
offset 0_15	0x0000ffff & address
offset 16_31	0x0000ffff & (address >>16)

Call-gate in Long Mode Call-gate are unavoidable parts of Intel structure, and even in 64-bit long mode. In addition to GDT, LDT is also present but special cases like segmentation using the FS/GS segment are replaced by the new MSR-based mechanism using *IA32_GS_BASE* and *IA32_KERNEL_GS_BASE* MSRs [7]. The fact that LDT and GDT are still presented in long mode is used in Windows when the kernel utilizes the UMS (User-Mode Scheduling). So Windows creates a Local Descriptor Table if a thread tends to use UMS [11].

3.5 Disabled UIMP

As described previously UIMP protection could be employed as an external privilege check. However, in our observations Linux and Windows do not use UIMP features for some compatibility issues. Thus, this opens a kernel memory address leak to user-mode applications, and valid addresses can be used for exploiting the Operating System Kernel or as a valid address for other side-channel measurements. In the following section, we demonstrate how these addresses could lead to a full system compromise. Nevertheless, Microsoft decided to remove the support for GDT, SIDT, SLDT, SMSW, and STR instructions in hypervisor as explained. Our observation shows that even if operating systems use UMIP or DESCRIPTOR-TABLE EXITING separately or both of them simultaneously, it is still vulnerable to side-channel attacks based on TSX.

3.6 Far Calls and Far JMPs

The far forms of JMP and CALL refer to other segments and require privilege checking. The far JMP and CALL can be performed in two methods:

- Without call-gate Descriptor: The processor permits a JMP or CALL directly to another segment only if:
  1. DPL of the target segment = CPL of the calling segment
  2. Confirming bit of the target code is set and DPL of the target segment $\leq$ CPL

  Note that Confirming Segment may be called from various privilege levels, but is executed at the privilege level of the calling procedure.
- With call-gate Descriptor: The far pointer of the control transfer instruction uses the selector part of the pointer and selects a gate. The selector and offset fields of a gate form a pointer to the entry of a procedure.

4 The Proposed Attack

In this section, we describe how the explored mechanism are used to create the attack. Then, we show the results obtained from the Intel processor and show how the valid base address of IDT and GDT could be obtained without using SIDT and SGDT. Next, we show how to build a valid call-gate entry and use it in combination with a write-what-where to execute an adversary code. Then attacker crafts the shellcode in *ring 0* in order to elevate privilege or hide the malware in the kernel. Figure 2 illustrates the high level overview of the proposed attack.

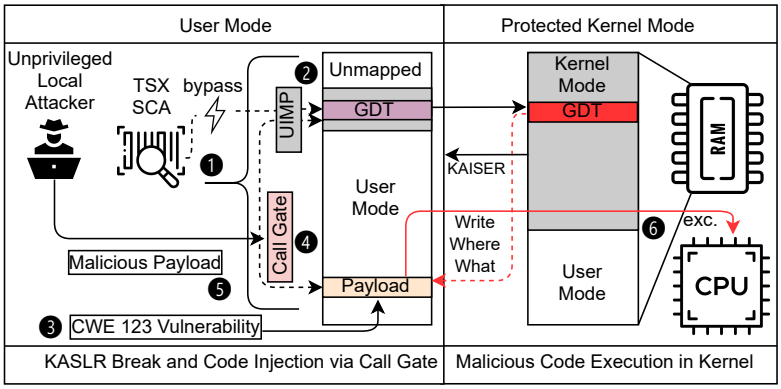

Fig. 2. A high-level overview of the proposed attack.

According to Figure 2, in ❶ the local unprivileged adversary carries out a conventional TSX timing side channel in order to disclose GDT address, bypassing UIMP ❷. The details is explained in Section 4.3. In ❸, the attacker arms the procedure with an existing Write Where What vulnerability explained in Section 4.7. Then in ❹, the adversary configures The GDT descriptions to point out his own malicious payload via the Call Gate features in accordance to the descriptions in Section 4.5. Finally the malicious data is loaded in the desired address in ❺ and wrongly executed by the processor with the proper permissions in ❻.

4.1 Threat Model

As a basic assumption for the attack model, the attacker can execute code in the victim's computer in a limited level of privilege, including a highly limited user-mode or in a sandboxed application with all the common defenses (e.g., SMEP, SMAP, DEP) enabled and configured suitably. In order to fully compromise the system an attacker has a prior *write-what-where* (CWE-123) [25] vulnerability in operating system kernel. Furthermore, as an extension to the proposed attack mechanism, the adversary might also execute code in a vitalized environment as well in the shared-computing platform (e.g. cloud computing) scenario.

4.2 Experimental Setup

The experiment to showcase the effectiveness of the attack chain has been executed on a system equipped with 9^{th} generation of Intel processor (i9-9880H), running on a Windows 19H1 (also known as 1903) with 16 GB of DDR4 RAM. Moreover, the same attack procedure is carried out on a system with a 6^{th} generation CPU (6820HQ), to ensure the generalization of the method. The test has also been successfully experimented on 19H2 and the latest 20H1 Microsoft Windows, Ubuntu Debian 7, and Mac OSX Mojave as well.

4.3 Finding GDT Address

In order to locate the GDT address, a timing measurement is required to discover the elapsed time in accessing a *mapped* and an *unmapped* address in the kernel space memory. Experimentally, a valid address gives the response time about $190 \sim 197$ clock-cycles (different based on architecture) and an invalid address access returns after about $220 \sim 234$ clock-cycles based on our results in 6^{th} Gen Intel (6820HQ).

To implement such a measurement, a combination of the kernel memory address and access time (RDTSCP) + TSX (XBEGIN, XEND) is employed. Then the response time difference in accessing a mapped and unmapped addresses could lead to the identification of mapped addresses.

Furthermore, if a particular processor does not support the RDTSCP instruction, then one could get similar results by the serialization process. More precisely, it is required to serialize instructions to execute all of the instructions fetched before the targeted instruction. So a combination of CPUID + RDTSC is adequately employed.

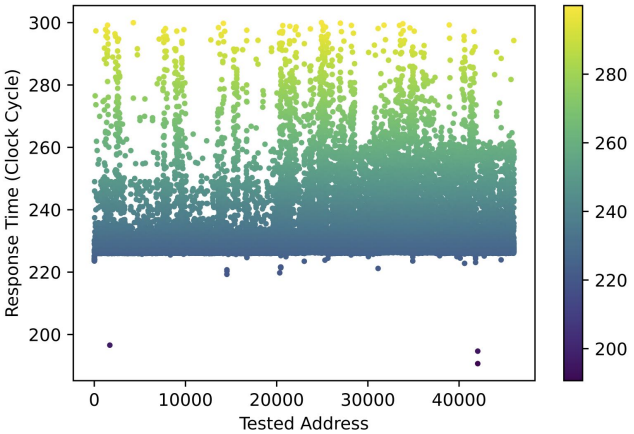

Fig. 3. The results of timing TSX-based measurements on a uni-core system

Note that the first implementation indeed gives more precise results compared to executing RDTSC. Our experiments show that it is not suitable to use CPUID for the second RDTSC as it takes several clocks-cycles. Also, it would be possible to use the timing thread, if a operating system prohibits the usage of RDTSC or RDTCSP [8], or intercepts the execution of CPUID using Intel VMX [14] or Intel FlexMigration [10]. Timing threads could even have a higher resolution rather than RDTSC/RDTCSP on many processors [31] [30]. By deploying these instructions, an automatic process is triggered to find valid targeted addresses.

```
1  rdtscp   ; get the current time clock of processor
2   ...        ; save the rdtscp results somewhere (e.g
     registers)
3
4  mov rax,[Kernel Address]  ; Move a kernel address into tax
5  xbegin  $+xxx     ; Use Intel TSX in order to suppress any
     error in user-mode
6  ; The error always happens because we are trying to read
     kernel address
7  mov byte ptr [rax], 0    ; Try to write into kernel address
8   ...       ; Error occurs here (program never reaches here)
9
10 xend     ; End of TSX
11 rdtscp    ; Compute the core clock timing again in order to
     see how many
12         ; clocks CPU spends when trying too write into our
     address
```

Listing 1.1. The timing measurement code deployed by the use of TSX technology (RDTSCP)

The result consists of four valid elements. The first one is the addresses that are valid for IDT. Second is the address of GDT, and third is the address of SYSCALL MSR_LSTAR (`0xC0000082`) - (The kernel's RIP SYSCALL entry for 64-bit software) [36]. Finally, the fourth is where the page tables are mapped. The timing results of the deployed measuring method is depicted in Figure 3.

Our observation in the latest 20H1 (and other versions of Windows) shows that GDT and IDT are mapped in a particular order, even though there is no limitation to assign different addresses. By way of example, Windows maps IDT in a unique address. IDTR is `fffff80021eeb000`, and GDTR `fffff80021eedfb0` (GDTR = GDT Base + GDT size) and this sequence is the same each time Windows is restarted when the KASLR addresses changed. The difference is `0x2000` bytes or two pages. Thus, the address of IDT could first be determined, leading to revealing the address of GDT where another page of `0x2000` bytes is mapped following the first valid page address.

While there are other pages mapped into memory addresses (e.g., shadow functions for system-calls and interrupts), the addresses are far from the target addresses (e.g., `fffff8001d34e500`). Therefore, the address among IDT, GDT, Interrupt Shadows, and System Call Shadows could be identified. A payload for

call-gate could build later finding the GDT base address. The following commands in Listing 1.2 shows the process of identification of valid addresses on all the cores by realizing the distance between GDT and IDT addresses.

```
; Accessing First Core's IDT and GDT
0: kd> r idtr
idtr=fffff8077925b000
0: kd> r gdtr
gdtr=fffff8077925dfb0
; Accessing Second Core's IDT and GDT
0: kd> ~1
1: kd> r idtr
idtr=ffff8401bc053000
1: kd> r gdtr
gdtr=ffff8401bc055fb0
; Accessing Third Core's IDT and GDT
1: kd> ~2
2: kd> r idtr
idtr=ffff8401bc0f5000
2: kd> r gdtr
gdtr=ffff8401bc0f7fb0
; Accessing Forth Core's IDT and GDT
2: kd> ~3
3: kd> r idtr
idtr=ffff8401bc1a4000
3: kd> r gdtr
gdtr=ffff8401bc1a6fb0
```

Listing 1.2. The procedure of employing IDTR and GDTR

We observed that allocated addresses for IDT and GDT have a special pattern for each core. For instance, here are several addresses that Windows allocated for IDT of its first core:

- `fffff8036385b000` , `fffff8027ca5b000`
- `fffff80053a5b000` , `fffff8076525b000`

Our experiments indicate that these addresses tend to follow a specific pattern. As the pseudo-code illustrated in Listing 1.2, the GDT has the same pattern As IDT as well. Our experiments show that, regardless of the system in hand, for the first core, the pattern of `fffff80XXXX5b000` is spotted, where `XXXX` can be changed due to the prevention mechanism of KASLR. The first bytes in the pattern address is to create a canonical address, and the least significant byte has a constant value of `5b000` pattern. This brings 0xffff = 65535 possibilities to find the address of IDT and GDT in the first core of Windows. The same pattern can be applied to other cores as well. In a uni-core system, one can test up to 10 addresses per second with excellent precision, using the explained timing side-channel. Moreover, one could also hasten this measurement up to 20 addresses per second, in compromise to the loss of accuracy. Approximately, it takes 109 minutes to find the address of the GDT for the first core. Of course, the patterns

for other cores could be discovered as well. As an example, in the 8-core system, there are eight possibilities for IDT and GDT addresses, which could speed up the search 8x faster. Also, it is possible to use other cores simultaneously for accelerating the search process.

4.4 Build call-gate Entry

We have built our payload based on the description discussed in section 3.4.

4.5 Using FAR JMPs, FAR CALLs

As explored in section 3.6, the near forms of JMP and CALL transfer within the current code segment requires only limited checking. However, the far forms of JMP and CALL are referred to as other segments and require privilege checking. Hence, when the CPU fetches a far-call instruction, it will use that instruction's 'selector' value to look up a descriptor in the GDT (or in the current LDT).

If the call-gate descriptor is fetched, and if access is allowed (i.e., if CPL $\leq$ DPL), then the CPU will perform a complex sequence of actions which will accomplish the requested ring-transition. CPL is based on the least significant 2-bits in register CS (also in SS). The new value for SS:SP comes from a special system-segment, known as the TSS (Task State Segment). The CPU locates its TSS by referring to the value in register TR (Task Register).

4.6 Returning back to the user-mode

After the call-gate is executed in kernel-mode, and we run shellcode in kernel-mode, it is time to return to the user-mode in order to avoid a crash in kernel-mode like BSOD in Windows or Kernel Panic in Linux.

In order to return to user-mode or any other outer ring that is used as the source of FAR CALL or FAR JMP, one should execute *lret* instruction in the inner ring. It is analogous to the procedure when an interrupt is returned to the previous state.

1. Use the far-return instruction: 'lret'
   - Restores CS:IP from the current stack , Restores SS:SP from the current stack
2. Use the far-return instruction: 'lret $n'
   - Restores CS:IP from the current stack
   - Discards n parameter-bytes from that stack , Restores SS:SP from that current stack

4.7 Combining attack with CWE-123

CWE-123 stands for write-what-where bugs. We have employed CVE-2016-7255 to modify our specific GDT entries. Consequently, the kernel-mode code execution of the shell-code using a FAR CALL is achieved. Also, another effect of this attack is to change the supervisor bit of page table so that page tables are readable and writable in user-mode or *self-ref of death attack).*

5 Discussion: The Possible Mitigation

The simple approach of complete isolation of the kernel is not able to fully unmap GDT from the user-mode since, in all modes of execution, the GDT descriptors should be available. Every segment register has a *visible* part and a *hidden* part. The hidden part sometimes referred to a *descriptor cache* or a *shadow register*. When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. The information cached in the segment register (visible and hidden) allows the processor to translate addresses without taking extra bus cycles to read the base address and limit from the segment descriptor. In systems in which multiple processors have access to the same descriptor tables, it is the responsibility of software to reload the segment registers when the descriptor tables are modified. Otherwise, an old segment descriptor cached in a segment register might be used after its memory-resident version has been modified [6].
In our mitigation setup we used a custom hypervisor to monitor and detect any *SYSCALL*, *SYSRET*, and interrupt execution. We deploy the proposed mitigation to switch GDT/IDT entries between user mode and kernel mode. Our hypervisor simulation shows a 2.7% delay overhead due to the additional execution introduced by the mitigation. However, the same methodology could be deployed within the operating system reducing the overhead significantly.

It is worthy of mentioning that, complete mitigation to this attack would be the employment of separate GDT base in kernel and user layout. The kernel GDT should not be mapped into the user-mode, and Operating System Kernel has to change the address of GDTR each time a ring modification occurs. For example, it shall use SGDT to change the GDTR after every user-mode to kernel-mode switch caused by *SYSENTER* and *SYSCALL* or every interrupts handler routines. The mapped GDT in the user-mode should also be modifiable only by the kernel (not user-mode). Hence, the user-mode application cannot access a valid address for GDT, and the discovered GDT address by the attacker is only valid when it is on user-mode. So, if a bug such as Write-What-Where occurs in the kernel or any system-level driver or kernel module, it cannot modify the user-mode GDT; thus, if the user-mode application tries to use call-gate in ring 3, the corresponding GDT entry is invalid, and the attack fails.

6 Related Efforts

Micro-architectural software attacks have been widely investigated in the context of revealing or damaging private and sensitive data. Recent works such as [3, 34, 37] aim to discover data on the victim system secretly. Recent works have demonstrated that the state of the art mitigation for such attack are still insufficient. Authors in [16], present a novel memory-sharing-based attack that breaks the KASLR on KPTI-enabled Linux virtual machines. Similarly, TagBleed [19], abuses tagged TLBs and residual translation information to break

KASLR. Furthermore, adversary techniques for exploitation on shared Virtual Environments like [27] have shown to be promising in practice.
With regards to much older timing side-channel attacks, Osvik et al. [28] introduced the PRIME+PROBE on the L1 cache, to attack the AES implementations, discovering secret keys. Consequently, more promising and sophisticated methods like [37] were proposed.

Moreover, other software-based attacks take advantage of on DRAM pioneered by [17] have also shown to be very practical, jeopardizing the private data stored in memory in various circumstances. In terms of exploiting the abandoned, but existing technologies in modern CPU designs, which is the primary concern of this paper, the possible vulnerabilities regarding the structure of GDT and IDT, were previously studied by [12]. Researchers in [12] proposed a technique to gain a more stable kernel-level exploitation. These techniques were shown to be applicable in Windows-NT systems. Moreover, interestingly, several utilized mechanisms in this article, such as call-gate has also been used for securing the systems. For instance, [21] present an approach to prevent sandbox leakage based on call-gate.

7 Conclusion

The impact of the hardware vulnerability exploited by software techniques has been proved to be dreadful. In this paper, we presented a TSX based side-channel attack, revealing the addresses of GDT and IDT in the kernel space, which could be exploited by an arbitrary user-mode application. We demonstrated that a single *Write-What-Where* vulnerability in the operating system could lead to a full system compromise through call-gate feature available in today's CPUs, irrespective of the version of the operating system. We have successfully evaluated our method by implementing an attack on the 9^{th} Generation *Intel* processors.

The attack presented here is based on the descriptor structures available on the modern processors (e.g., Intel as well as AMD [1]) although have hidden address by ASLR but are mapped into the user-mode address layout. The exploitation perfectly works with common *Write What Where* bugs. For instance, any bug in a *JavaScript* application on an isolated web-browser in the kernel address or graphic functions of the operating system (e.g., Win32k bugs in Windows) will be enough to be exploited. Moreover, we suggested software mitigation for this vulnerability since the presented attack bypasses the recent mitigation to Meltdown Attack (e.g., KAISER).

References

1. Devices, A.M.: Amd64 architecture programmer's manual volume 2: System programming (2006)
2. Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering **8**(1), 1–27 (2018)

3. Gras, B., Razavi, K., Bos, H., Giuffrida, C.: Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 955–972 (2018)
4. Gruss, D., Hansen, D., Gregg, B.: Kernel isolation: From an academic idea to an efficient patch for every computer. login: USENIX Magazine **43**(4), 10–14 (2018)
5. Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., Mangard, S.: Kaslr is dead: long live kaslr. In: International Symposium on Engineering Secure Software and Systems. pp. 161–176. Springer (2017)
6. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 3C: Chapter 24, VIRTUAL MACHINE CONTROL STRUCTURES (Table 24-6. Definitions of Primary Processor-Based VM-Execution Controls) **3C** (2019)
7. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 4: Chapter 2, MODEL-SPECIFIC REGISTERS (MSRS) (Table 2-2. IA-32 Architectural MSRs) **4** (2019)
8. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 3A: Chapter 1, SYSTEM ARCHITECTURE OVERVIEW, Time Stamp Disable) **3A** (2019)
9. Hajihassani, O., Monfared, S.K., Khasteh, S.H., Gorgin, S.: Fast aes implementation: A high-throughput bitsliced approach. IEEE Transactions on Parallel and Distributed Systems **30**(10), 2211–2222 (2019)
10. Intel: Intel virtualization technology flexmigration application note (2012), `https://www.intel.com/content/dam/www/public/us/en/documents/application-notes/virtualization-technology-flexmigration-application-note.pdf`
11. Ionescu, A.: blog post (2018), `http://www.alex-ionescu.com/?p=340`
12. Jurczyk, M., Coldwind, G.: Gdt and ldt in windows kernel vulnerability exploitation (2010)
13. Karvandi, S.: Call gates' ring transitioning in ia-32 mode (2019), `https://rayanfam.com/topics/call-gates-ring-transitioning-in-ia-32-mode/`
14. Karvandi, S.: Hypervisor from scratch – part 6: Virtualizing an already running system (2019), `https://rayanfam.com/topics/hypervisor-from-scratch-part-6/`
15. Kiarostami, M.S., Reza Daneshvaramoli, M., Monfared, S.K., Rahmati, D., Gorgin, S.: Multi-agent non-overlapping pathfinding with monte-carlo tree search. In: 2019 IEEE Conference on Games (CoG). pp. 1–4 (2019)
16. Kim, T., Kim, T., Shin, Y.: Breaking kaslr using memory deduplication in virtualized environments. Electronics **10**(17), 2174 (2021)
17. Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O.: Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. In: ACM SIGARCH Computer Architecture News. vol. 42, pp. 361–372. IEEE Press (2014)
18. Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. arXiv preprint arXiv:1801.01203 (2018)
19. Koschel, J., Giuffrida, C., Bos, H., Razavi, K.: Tagbleed: Breaking kaslr on the isolated kernel address space using tagged tlbs. In: 2020 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 309–321. IEEE (2020)
20. Kurth, M., Gras, B., Andriesse, D., Giuffrida, C., Bos, H., Razavi, K.: NetCAT: Practical Cache Attacks from the Network. In: S&P (May 2020), `https://www.vusec.net/download/?t=papers/netcat_sp20.pdf`, intel Bounty Reward
21. Lewis, P.: Using a call gate to prevent secure sandbox leakage (Sep 3 2013), uS Patent 8,528,083

22. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., et al.: Meltdown: Reading kernel memory from user space. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 973–990 (2018)
23. (Microsoft), S.: Kva shadow: Mitigating meltdown on windows (2018), https://msrc-blog.microsoft.com/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/
24. Minkin, M., Moghimi, D., Lipp, M., Schwarz, M., Van Bulck, J., Genkin, D., Gruss, D., Piessens, F., Sunar, B., Yarom, Y.: Fallout: Reading kernel writes from user space. arXiv preprint arXiv:1905.12701 (2019)
25. MITRE: Cwe-123: Write-what-where condition (2019), https://cwe.mitre.org/data/definitions/123.html
26. Monfared, S.K., Hajihassani, O., Kiarostami, M.S., Zanjani, S.M., Rahmati, D., Gorgin, S.: Bsrng: A high throughput parallel bitsliced approach for random number generators. In: 49th International Conference on Parallel Processing-ICPP: Workshops. pp. 1–10 (2020)
27. Oliverio, M., Razavi, K., Bos, H., Giuffrida, C.: Secure page fusion with vusion: https://www. vusec. net/projects/vusion. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 531–545 (2017)
28. Osvik, D.A., Shamir, A., Tromer, E.: Cache attacks and countermeasures: the case of aes. In: Cryptographers' track at the RSA conference. pp. 1–20. Springer (2006)
29. Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., Gruss, D.: Zombieload: Cross-privilege-boundary data sampling. arXiv preprint arXiv:1905.05726 (2019)
30. Schwarz, M., Maurice, C., Gruss, D., Mangard, S.: Fantastic timers and where to find them: high-resolution microarchitectural attacks in javascript. In: Int. Conf. on Financial Cryptography and Data Security. pp. 247–267. Springer (2017)
31. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: Using sgx to conceal cache attacks. In: Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 3–24. Springer (2017)
32. Seaborn, M., Dullien, T.: Exploiting the dram rowhammer bug to gain kernel privileges. Black Hat **15** (2015)
33. Stecklina, J., Prescher, T.: Lazyfp: Leaking fpu register state using microarchitectural side-channels. arXiv preprint arXiv:1806.07480 (2018)
34. Van Schaik, S., Giuffrida, C., Bos, H., Razavi, K.: Malicious management unit: Why stopping cache attacks in software is harder than you think. In: 27th {USENIX} Security Symposium ({USENIX} Security 18). pp. 937–954 (2018)
35. Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution (2018)
36. Wiki, O.D.: Sysenter (2017), https://wiki.osdev.org/SYSENTER
37. Yarom, Y., Falkner, K.: Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. In: 23rd {USENIX} Security Symposium ({USENIX} Security 14). pp. 719–732 (2014)