Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Werner Haas[2], Jann Horn[3], Stefan Mangard[1], Paul Kocher[4], Daniel Genkin[5], Yuval Yarom[6], Mike Hamburg[7], Raoul Strackx[8]
[1]Graz University of Technology, [2]Cyberus Technology GmbH, [3]Google Project Zero, [4]Independent (www.paulkocher.com), [5]University of Michigan, [6]University of Adelaide & Data61, [7]Rambus, Cryptography Research Division, [8]imec-DistriNet, KU Leuven

1 Introduction

Memory isolation is a cornerstone security feature in the construction of every modern computer system. Allowing the simultaneous execution of multiple mutually-distrusting applications at the same time on the same hardware, it is the basis of enabling secure execution of multiple processes on the same machine or in the cloud. The operating system is in charge of enforcing this isolation, as well as isolating its own kernel memory regions from other users.

Given its central role, on modern processors, the isolation between the kernel and user processes is backed by the hardware, in the form of a *supervisor bit* that determines whether code in the current context can access memory pages of the kernel. The basic idea is that this bit is set only when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process, thus supporting very efficient transitions from the user process to the kernel (e.g., for interrupt handling) while maintaining the security of the kernel memory space.

In this work we present Meltdown, a novel attack that exploits a vulnerability in the way the processor enforces memory isolation.

Root Cause. At a high level, the root cause of Meltdown's simplicity and strength are side effects caused by *out-of-order execution*, which is an important performance feature of modern processors designed to overcome latencies of busy execution units (e.g., a memory fetch unit waiting for data arrival from memory). Rather than stalling the execution, modern processors run operations *out-of-order*, *i.e.*, they look ahead and schedule subsequent operations on available execution units of the core.

While this feature is beneficial for performance, from a security perspective, one observation is particularly significant. Some CPUs allow an unprivileged process to load data from a privileged (kernel or physical) address into a temporary register, delaying exception handling to later stages. The CPU even allows performing further computations based on this register value, such as using it as an index to an array access. When the CPU finally realizes the error, it reverts the results of this incorrect transient execution, discarding any modifications to the program state (e.g., registers). However, we observe that out-of-order memory lookups influence the internal state of the processor, which in turn can be detected by the program. As a result, an attacker can dump the entire kernel memory by reading privileged memory in an out-of-order execution stream, and subsequently transmitting the data via a covert channel, e.g., by modulating the state of the cache. As the CPU's internal state is not fully reverted, the receiving end of the covert channel can later recover the transmitted value, e.g., by probing the state of the cache.

Conference'17, July 2017, Washington, DC, USA
2020. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Arch.	Description
x86	Most Intel and VIA processors are vulnerable. AMD processors are not.
Arm	Cortex-A75 and SoCs based on it are vulnerable. Some proprietary Arm-based processors, including some Apple and Samsung cores, are also vulnerable. Arm Cortex-A72, Cortex-A57 and Cortex-A15 are vulnerable to a Variant 3a of Meltdown. Other Arm cores are not known to be vulnerable.
Power	All IBM Power architecture processors are vulnerable.
z/Arch.	IBM z10, z13, z14, z196, zEC12 are vulnerable.
SPARC	V9-based SPARC systems are not vulnerable. Older SPARC processors may be impacted.
Itanium	Itanium processors are not vulnerable.

Table 1: Summary of processors affected by Meltdown.

Threat Model. To mount Meltdown, the adversary needs the ability to execute code on a vulnerable machine. Executing code can be achieved through various means, including hosting in cloud services, apps in mobile phones, and JavaScript code in web sites. Vulnerable machines include personal computers and servers featuring a large range of processors (see Table 1). Furthermore, while countermeasures have been introduced to both operating systems and browsers, these only became available after the disclosure of Meltdown.

Impact. Three properties of Meltdown combine to have a devastating effect on the security of affected systems. First, exploiting a hardware vulnerability means that the attack does not depend on specific vulnerabilities in the software. Thus, the attack is generic and, at the time of discovery, affected all existing versions of all major operating systems. Second, because the attack only depends on the hardware, traditional software-based protections, such as cryptography, operating system authorization mechanisms, or antivirus software, are powerless to stop the attack. Last, because the vulnerability is in the hardware, fixing the vulnerability requires replacing the hardware. While software-based countermeasures for Meltdown have been developed, these basically avoid using the vulnerable hardware feature, incurring a significant performance hit.

Evaluation. We evaluate the attack on modern desktop machines and laptops, as well as servers and clouds. Meltdown is effective against all major operating systems (including Linux, Android, OS X and Windows), allowing an unprivileged attacker to dump large parts of the physical memory. As the physical memory is shared among all other tenants running on the same hardware, this may include the physical memory of other processes, the kernel, and in the case of paravirtualization, the memory of the hypervisor or other co-located instances. While the performance heavily depends on the specific machine, e.g., processor speed, TLB and cache sizes, and DRAM speed, we can dump arbitrary kernel and physical memory at a speed of 3.2 KiB/s to 503 KiB/s.

Countermeasures. While not originally intended to be a countermeasure for Meltdown, KAISER [3], developed initially to prevent side-channel attacks targeting KASLR, also protects against Meltdown. Our evaluation shows that KAISER prevents Meltdown to a large extent. Consequently, we stress that it is of utmost importance to deploy KAISER on all operating systems immediately. Fortunately, during the responsible disclosure window, the three major operating systems (Windows, Linux, and OS X) implemented variants of KAISER and recently rolled out these patches.

Spectre Attacks and Followup Works. Meltdown was published simultaneously with the Spectre Attack [14], which exploits a different CPU performance feature, called *speculative execution*, to leak confidential information. Meltdown is distinct from Spectre in several ways, notably that Spectre requires tailoring to the victim process's software environment, but applies more broadly to CPUs and is not mitigated by KAISER. Since the publication of Meltdown and Spectre, several prominent follow-up works exploited out-of-order and speculative execution mechanisms to leak information across other security domains [11, 13, 15, 18, 20–22]. While some of these attacks have been mitigated, additional work is required to mitigate others.

2 Background

In this section, we provide background on out-of-order execution, address translation, and cache attacks.

2.1 The Microarchitecture

The Instruction Set Architecture (ISA) of a processor is the interface it provides to the software it executes. The ISA is typically defined as some state, which mostly consists of the contents of the architectural registers and the memory, and a set of instructions that operate on this state. The implementation of this interface consists of multiple components, collectively called the *microarchitecture*. The microarchitecture maintains a state which extends the architectural state of the processor as defined by the ISA, adding further information required for the operation of the microarchitectural components. While changes in the microarchitectural state do not affect the logical behavior of the program, they may affect its performance. Thus, the microarchitectural state of the processor depends on prior software execution and affects its future behavior, creating the potential for untraditional communication channels [2].

2.2 Out-of-order execution

Out-of-order execution [7] is an optimization technique that increases the utilization of the execution units of a CPU core. Instead of processing instructions strictly in sequential program order, waiting for slow instructions to complete before executing subsequent instructions, the CPU executes them as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. Hence, instructions execute in parallel as long as their results follow the architectural definition.

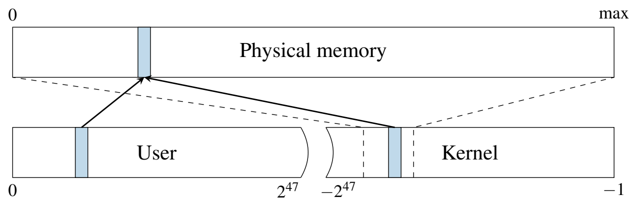

Figure 1: On Unix-like 64-bit systems, a physical address (blue) which is mapped accessible to the user space is also mapped in the kernel space through the direct mapping.

2.3 Address Spaces

To isolate processes from each other, CPUs support virtual address spaces where virtual addresses are translated to physical addresses. The operating system kernel plays a key role in managing the address translation for processes. Consequently, the memory of the kernel need also be protected from user processes. Traditionally, in segmented architectures [7], the kernel had its own segments which were not accessible to user processes.

In modern processors, a virtual address space is divided into a set of pages that can be individually mapped to physical memory through a multi-level page translation table. In addition to the virtual to physical mapping, the translation tables also specify protection properties that specify the allowed access to the mapped pages. These properties determine, for example, whether pages are readable, writable, and executable. A pointer to the currently used translation table is held in a dedicated CPU register. During a context switch, the operating system updates this register to point to the translation table of the next process, thereby implementing a per-process virtual address space, allowing each process to only reference data that belongs to its virtual address space. To reduce the cost of consulting the translation tables, the processor caches recent translation results in the Translation Lookaside Buffer (TLB).

While the TLB reduces the cost of address translation, its contents needs to be flushed when changing address space. To avoid the cost of flushing the TLB on every switch between the user program and the kernel, modern systems include the kernel memory within the address space of every process. Following an idea first introduced in the VAX/VMS system, the page table also includes a protection bit that indicates whether the page belongs to the kernel or to the user program. Kernel pages are only accessible when the processor executes with high privileges, *i.e.*, when executing the kernel. Thus, user processes are not able to read or to modify the contents of the kernel memory.

To aid in memory management, many modern operating systems directly map (a part of) the physical memory in the kernel's part of the virtual address space at a fixed location m (c.f. Figure 1). A physical address p can than be assessed through virtual address $p + m$.

2.4 Cache Attacks

To speed-up memory accesses and address translation, the CPU contains small memory buffers, called caches, that store frequently used data. CPU caches hide slow memory accesses by buffering frequently used data in smaller and faster internal memory. Modern CPUs have multiple levels of caches that are either private per core or shared among them. Address space translation tables are also stored in memory and, thus, also cached in the regular caches.

Cache side-channel attacks exploit the timing differences that the caches introduce. Several cache attack techniques have been proposed and demonstrated in the past, including Prime+Probe [16] and Flush+Reload [23]. Flush+Reload attacks work on a single cache line granularity. These attacks exploit the shared, inclusive last-level cache. An attacker frequently flushes a targeted memory location using the `clflush` instruction. By measuring the time it takes to reload the data, the attacker determines whether the memory location was loaded into the cache by another process since the last `clflush`. The Flush+Reload attack has been used for attacks on various computations, e.g., cryptographic algorithms [23], web server function calls [24], user input [6], and kernel addressing information [4].

A special use case of a side-channel attack is a covert channel. Here the attacker controls both the part that induces the side effect, and the part that measures the side effect. This can be used to leak information from one security domain to another while bypassing any boundaries existing on the architectural level or above. Both Prime+Probe and Flush+Reload have been used in high-performance covert channels [5].

3 A Toy Example

In this section, we start with a toy example, which illustrates that out-of-order execution can change the microarchitectural state in a way that leaks information.

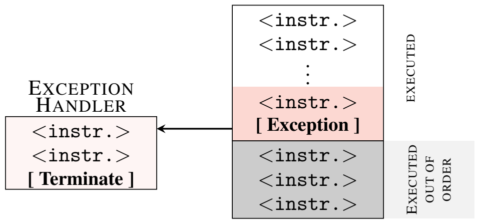

Figure 2: If an executed instruction causes an exception, control flow is diverted to an exception handler. Subsequent instruction may already have been partially executed, but not retired. Architectural effects of this transient execution are discarded.

Triggering Out-of-Order Execution. Figure 2 shows a simple code sequence first raising an (unhandled) exception and then accessing an array. The exception can be raised through any mean, such as accessing an invalid memory address, performing a privileged instruction in user code, or even division by zero. An important property of an exception, irrespective of its cause, is that the control flow does not follow program order to the code following the exception. Instead, it jumps to an exception handler in the operating system kernel. Thus, the code in our toy example is expected not to access the array because the exception traps to the kernel and terminates the application before the access is performed. However, we note that the access instruction after the exception has no data dependency on the trapping instruction. Hence, due to out-of-order execution, the CPU might execute the access before triggering the exception. When the exception is triggered, instructions executed out of order are not retired and, thus, never have architectural effects. However, instructions executed out-of-order do have side-effects on the microarchitecture. In particular, the contents of the memory accessed after the exception in Figure 2 are fetched into a register and also stored in the cache. When the out-of-order execution is reverted (*i.e.*, the register and memory contents are never committed), the cached memory contents survive reversion and remain in the cache for some time. We can now leverage a microarchitectural side-channel attack, such as Flush+Reload [23], to detect whether a specific memory location is cached, thereby making the affected microarchitectural state visible.

Observing Out-of-Order Execution. The code in Figure 2 accesses a memory address that depends on the value of `data`. As `data` is multiplied by 4096, data accesses to `array` are scattered over the array with a distance of 4 KiB (assuming a 1 B data type for `array`). Thus, there is an injective mapping from the value of `data` to a memory page, *i.e.*, different values for data never result in accesses to the same page. Consequently, if a cache line of a page is cached, we can determine the value of `data`.

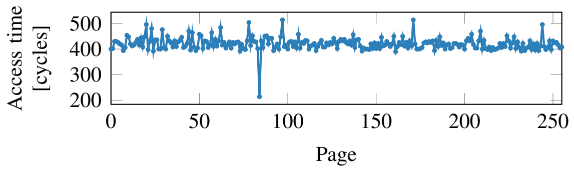

Figure 3: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of `array` shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Figure 3 shows the result of Flush+Reload measurements iterating over all of the pages of `array`, after executing the out-of-order snippet in Figure 2 with `data = 84`. Although the array access should not have happened due to the exception, we can clearly see that the index which would have been accessed is cached. Iterating over all pages (e.g., in the exception handler) shows a cache hit for page 84 only. This demonstrates that instructions which are only executed out-of-order but are never retired, change the microarchitectural state of the CPU. In Section 4 we show how we modify this toy example to leak an inaccessible secret.

4 Building Blocks of the Attack

The toy example in Section 3 illustrates that side effects of out-of-order execution can modify the microarchitectural state to leak information. While the code snippet reveals the data value passed to a cache side channel, we want to show how this technique can be leveraged to leak otherwise inaccessible secrets. In this section, we want to generalize and discuss the necessary building blocks to exploit out-of-order execution for an attack.

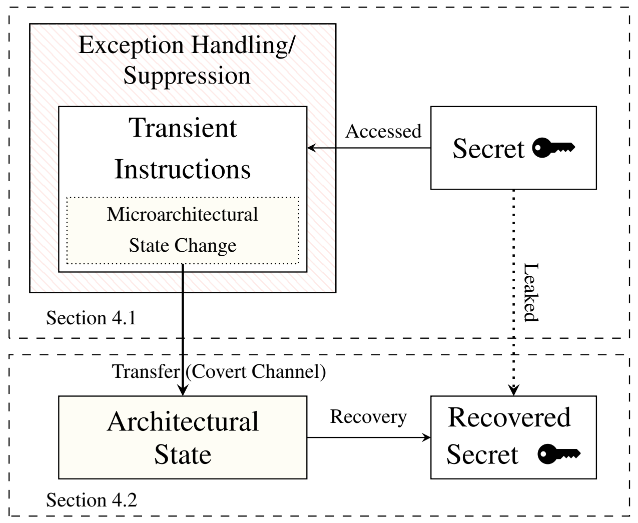

Figure 4: The Meltdown attack uses exception handling or suppression, e.g., TSX, to run a series of transient instructions. These transient instructions obtain a (persistent) secret value and change the microarchitectural state of the processor based on this secret value. This forms the sending part of a microarchitectural covert channel. The receiving side reads the microarchitectural state, lifts it to architectural, and recovers the secret value.

Overview Of Meltdown. Assume an adversary that targets a secret value which is kept somewhere in physical memory. The full Meltdown attack leaks this value using two building blocks, as illustrated in Figure 4. The first building block of Meltdown (Section 4.1) is to make the CPU execute one or more *transient* instructions, *i.e.*, instructions that do not occur during regular execution. The second building block of Meltdown is to transfer the microarchitectural side effect of the transient instruction sequence to an architectural state to further process the leaked secret. Thus, Section 4.2 describes methods to lift a microarchitectural side effect to an architectural state using covert channels.

4.1 Executing Transient Instructions

The first building block of Meltdown is the execution of transient instructions, which are executed out-of-order and leave measurable side effects. We focus on transient instructions that follow an illegal access to addresses that are mapped within the attacker's process such as user-inaccessible kernel space addresses. In general, accessing such user-inaccessible addresses triggers an exception, which typically terminates the application. Because we want to measure the microarchitectural state of the processor after the transient execution, we want to avoid terminating the process. We now present several approaches the attacker can use to cope with the exception.

Fork-and-Crash. A trivial approach is to fork the attacking application before accessing the invalid memory location that terminates the process and only access the invalid memory location in the child process. The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by probing the microarchitectural state.

Exception Handling. Next, it is also possible to install a signal handler that is executed when a certain exception occurs, e.g., a segmentation violation. This allows the attacker to issue the instruction sequence and prevent the application from crashing, reducing the overhead as no new process has to be created.

Exception Suppression via TSX. An alternative approach to deal with exceptions is to prevent them from being raised in the first place. Intel Transactional Synchronization Extensions (TSX) defines the concept of *transaction*, which is a sequence of instructions that execute atomically, *i.e.*, either all of the instructions in a transaction are executed, or none of them is. If an instruction within the transaction fails, already executed instructions are reverted, but no exception is raised. By wrapping the code of Listing 1 in such a TSX transaction, the exception is suppressed. Yet, the microarchitectural effects of transient execution are still visible. Because suppressing the exception is significantly faster than trapping into the kernel for handling the exception, and continuing afterwards, this results in a higher channel capacity.

Exception Suppression via Branch Predictor. Finally, speculative execution issues instructions that might not occur in the program order due to a branch misprediction. Thus by forcing a misprediction that speculatively executes the invalid memory access, we can achieve transient execution of *both* the invalid memory access and the instructions following it, without triggering an exception. See Kocher et al. [14] for further details on speculative execution and transient instructions.

4.2 Building a Covert Channel

The second building block of Meltdown is lifting the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state (cf. Figure 4). The transient instruction sequence can be seen as the transmitting end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. Note that the receiver is not part of the transient instruction sequence and can be a different thread or process e.g., the parent process in the fork-and-crash approach.

A Cache-Based Covert Channel. Previous works [5, 16, 23] have demonstrated that the microarchitectural state of the cache can be easily lifted into a architectural state. We, therefore, employ these techniques for our covert channel. Specifically, we use Flush+Reload [23], as it allows building a fast and low-noise covert channel.

After accessing a user-inaccessible secret value, the transient instruction sequence executes the cache covert channel transmitter, performing a memory access using the secret value as part of the address. As explained earlier, this address remains cached for subsequent accesses, and survives the soon-to-be-raised exception. Thus, part of the cache state depends on the secret value, and lifting this state to an architectural state leaks the secret value.

Recovering the Leaked Value. The covert channel receiver can then monitor whether an address has been loaded into the cache by measuring the access time to the address. For example, the sender can transmit a '1'-bit by accessing an address which is loaded into the monitored cache, and a '0'-bit by not accessing such an address. Using multiple different cache lines, as in our toy example

```
1 mov al, byte [rcx] ; rcx = kernel address
2 shl rax, 0xc
3 mov rbx, qword [rbx + rax] ; rbx = probe array
```

Listing 1: The core of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. Subsequent instructions are executed out of order before the exception is raised, leaking the data from the kernel address through the indirect memory access.

in Section 3, allows transmitting multiple bits at once. For every one of the 256 different byte values, the sender accesses a different cache line. By performing a Flush+Reload attack on all of the 256 possible cache lines, the receiver can recover a full byte rather than just one bit of secret value. However, since the Flush+Reload attack takes much longer (typically several hundred cycles) than the transient instruction sequence, transmitting only a single bit at once is more efficient. The attacker can choose the bit to transmit by shifting and masking the secret value accordingly.

Using Other Covert Channels. Note that the covert channel part is not limited to cache-based microarchitectural channels. Any instruction (or sequence) that influences the microarchitectural state of the CPU in a way that can be observed from a user process can be used to build a covert channel transmitter. For example, to send a '1'-bit the sender could issue an instruction (or sequence) which occupies a certain execution port such as the ALU. The receiver measures the latency when executing an instruction (sequence) on the same execution port. A high latency implies that the sender sends a '1'-bit, whereas a low latency implies that the sender sends a '0'-bit. The advantage of the Flush+Reload cache covert channel is the noise resistance and the high transmission rate [5]. Furthermore, with cache architectures commonly used in current CPUs, different memory access latencies can be observed from any CPU core [23], *i.e.*, rescheduling events do not significantly affect the covert channel.

5 The Meltdown Attack

In this section, we present Meltdown, a powerful attack enabling arbitrary kernel memory (typically including the entire physical memory) to be read from an unprivileged user program, comprised of the building blocks presented in Section 4. First, we discuss the attack setting to emphasize the wide applicability of this attack. Second, we present an attack overview, showing how Meltdown can be mounted on both Windows and Linux on personal computers, on Android on mobile phones as well as in the cloud. Finally, we discuss a concrete implementation of Meltdown allowing to dump memory with 3.2 KiB/s to 503 KiB/s.

Attack Setting. In our attack, we consider personal computers and virtual machines in the cloud. In the attack scenario, the attacker can execute arbitrary unprivileged code on the attacked system, *i.e.*, the attacker can run any code with the privileges of a normal user. The attacker targets secret user data, e.g., passwords and private keys, or any other valuable information. Finally, we assume a completely bug-free operating system. That is, we assume that the attacker does not exploit any software vulnerability to gain kernel privileges or to leak information.

Attack Description. Meltdown combines the two building blocks discussed in Section 4. At a high level, Meltdown consists of 3 steps:

- **Step 1: Reading the Secret.** The content of an attacker-chosen memory location, which is inaccessible to the attacker, is loaded into a register.
- **Step 2: Transmit the Secret.** A transient instruction accesses a cache line based on the secret content of the register.
- **Step 3: Receive the Secret.** The attacker uses Flush+Reload to determine the accessed cache line and hence the secret stored at the chosen memory location.

By repeating these steps for different memory locations, the attacker can dump the kernel memory, including the entire physical memory.

Listing 1 shows a typical implementation of the transient instruction sequence and the sending part of the covert channel, using x86 assembly instructions. Note that this part of the attack could also be implemented entirely in higher level languages such as C. In the following, we discuss each step of Meltdown and the corresponding code line in Listing 1.

5.1 Step 1: Reading the Secret.

Recall that modern operating systems map the kernel into the virtual address space of every process. Consequently, a user process can specify addresses that map to the kernel space. As discussed in Section 2.3, in parallel with performing the access, the CPU verifies that the process has the required permission for accessing the address, raising an exception if the user tries to reference a kernel address. However, when translating kernel addresses they do lead to valid physical addresses, which the CPU can access, and only the imminent exception due to illegal access protects the contents of the kernel space. Meltdown exploits the out-of-order execution feature of modern CPUs, which execute instructions for a small time window between the illegal memory access and the subsequent exception.

Line 1 of Listing 1 loads a byte value from the target kernel address, pointed to by the `RCX` register, into the least significant byte of the `RAX` register represented by `AL`. The CPU executes this by fetching the `MOV` instruction, decoding and executing it, and sending it to the reorder buffer for retirement. As part of the execution, a temporary physical register is allocated for the updated value of architectural register `RAX`. Trying to utilize the pipeline as much as possible, subsequent instructions (Lines 2–3) are decoded and sent to the reservation station holding the instructions while they wait to be executed by the corresponding execution units.

Thus, when the kernel address is accessed in Line 1, it is likely that the CPU already issues the subsequent instructions as part of the out-of-order execution, and that these instruction wait in the reservation station for the content of the kernel address to arrive. When this contents arrives, the instructions can begin their execution. Furthermore, processor interconnects [10] and cache coherence protocols [19] guarantee that the most recent value of a memory address is read, regardless of the storage location in a multi-core or multi-CPU system.

When the processor finishes executing the instructions, they retire in-order, and their results are committed to the architectural state by updating the register renaming tables, *i.e.*, the mapping of architectural to physical registers [7]. During the retirement, any interrupts and exceptions that occurred while executing of the instruction are handled. Thus, when the `MOV` instruction that loads the kernel address (Line 1) is retired, the exception is registered, and the pipeline is flushed to eliminate all results of subsequent instructions which were executed out of order. However, there is a race condition between raising this exception and our attack Step 2 as described below.

5.2 Step 2: Transmitting the Secret

The instruction sequence from Step 1, which is executed out-of-order, is chosen such that it becomes a transient instruction sequence. If this transient instruction sequence is executed before the `MOV` instruction is retired, and the transient instruction sequence performs computations based on the secret, it can transmit the secret to the attacker.

As already discussed, we use cache attacks that allow building fast and low-noise covert channels using the CPU's cache. Thus, the transient instruction sequence has to encode the secret as a microarchitectural cache state, similar to the toy example in Section 3.

We allocate a probe array in memory and ensure that no part of this array is cached. To transmit the secret, the transient instruction sequence performs an indirect memory access to an address which depends on the secret (inaccessible) value. Line 2 of Listing 1, shifts the secret value from Step 1 by 12 bits to the left, effectively multiplying it by the page size of 4 KiB. This ensures that accesses to the array have a large spatial distance from each other, preventing the hardware prefetcher from loading adjacent memory locations into the cache. Here, we read a single byte at once. Hence, our probe array is 256×4096 bytes long.

Line 3 adds the multiplied secret to the base address of the probe array, forming the target address of the covert channel. It then accesses this address, effectively bringing its content to the cache. Consequently, our transient instruction sequence affects the cache state based on the secret value that was read in Step 1.

Finally, since the transient instruction sequence in Step 2 races against raising the exception, reducing the runtime of Step 2 can significantly improve the performance of the attack. For instance, taking care that the address translation for the probe array is cached in the Translation Lookaside Buffer (TLB) increases the attack performance on some systems.

5.3 Step 3: Receiving the Secret

In Step 3, the attacker recovers the secret value from Step 1 by implementing the receiving end of a microarchitectural covert channel that transfers the cache state (Step 2) back into an architectural state. As discussed in Section 4.2, our implementation of Meltdown relies on Flush+Reload for this purpose.

When the transient instruction sequence of Step 2 is executed, exactly one cache line of the probe array is cached. The position of the cached cache line within the probe array depends only on the secret, read in Step 1. To recover the secret, the attacker iterates over all 256 pages of the probe array and measures the access time to the first cache line of each page. The number of the page containing the cached cache line corresponds directly to the secret value.

5.4 Dumping Physical Memory

Repeating all three steps of Meltdown, an attacker can dump the entire memory by iterating over all addresses. However, as the memory access to the kernel address raises an exception that terminates the program, we use one of the methods from Section 4.1 to handle or suppress the exception.

Furthermore, because most major operating systems also map the entire physical memory into the kernel address space (cf. Section 2.3) in every user process, Meltdown can also read the entire physical memory of the target machine.

6 Evaluation

In this section, we evaluate Meltdown and the performance of our proof-of-concept implementation.[1] Section 6.1 discusses the information Meltdown can leak, and Section 6.2 evaluates the performance of Meltdown, including countermeasures. Our results are consistent across vulnerable laptops, desktop PCs, mobile phones, and cloud systems.

6.1 Leakage and Environments

We evaluated Meltdown on various operating systems with and without patches. On all unpatched operating systems, Meltdown can successfully leak kernel memory. We detail the Linux, Windows and Android evaluation here.

Linux. We successfully evaluated Meltdown on multiple versions of the Linux kernel, from 2.6.32 to 4.13.0, without the patches introducing the KAISER mechanism. On all of these Linux kernel versions, the kernel is mapped into the address space of user processes, but access is prevented by the permission settings for these addresses. As Meltdown bypasses these permission settings, an attacker can leak the complete kernel memory if the virtual address of the kernel base is known. Since all major operating systems (even 32 bit as far as possible) also map the entire physical memory into the kernel address space (cf. Section 2.3), all physical memory can also be read.

Before kernel 4.12, kernel address space layout randomization (KASLR) was not enabled by default [17]. Without KASLR, the entire physical memory was directly mapped starting at at address `0xffff 8800 0000 0000`. On such systems, an attacker can use Meltdown to dump the entire physical memory, simply by reading from virtual addresses starting at `0xffff 8800 0000 0000`. When KASLR is enabled, Meltdown can still find the kernel by searching through the address space. An attacker can also de-randomize the direct physical map by iterating through the virtual address space.

On newer systems KASLR is usually active by default. Due to the large size and the linearity of the mapping the randomization of the direct physical map is usually 7 bits or lower. Hence, the attacker can test addresses in steps of 8 GB, resulting in a maximum of 128 memory locations to test. Starting from one discovered location, the attacker can again dump the entire physical memory.

Windows. We successfully evaluated Meltdown on a recent Microsoft Windows 10 operating system, last updated just before patches against Meltdown were rolled out. In line with the results

[1] https://github.com/IAIK/meltdown

on Linux, Meltdown can leak arbitrary kernel memory on Windows. This is not surprising, since Meltdown does not exploit any software issues, but is caused by a hardware issue.

In contrast to Linux, Windows does not map the physical memory directly in the kernel's virtual address space. Instead, a large fraction of the physical memory is mapped in the paged pools, non-paged pools, and the system cache. Windows does map the kernel into the address space of every application. Thus, Meltdown can read kernel memory which is mapped in the kernel address space, *i.e.*, any part of the kernel which is not swapped out, and any page mapped in the paged and non-paged pool, and in the system cache.

Note that there are physical pages which are mapped in one process but not in the (kernel) address space of another process. These physical pages cannot be attacked using Meltdown. However, most of the physical memory is accessible through Meltdown.

We could read the binary code of the Windows kernel using Meltdown. To verify that the leaked data is indeed kernel memory, we first used the Windows kernel debugger to obtain kernel addresses containing actual data. After leaking the data, we again used the Windows kernel debugger to compare the leaked data with the actual memory content, confirming that Meltdown can successfully leak kernel memory.

Android. We successfully evaluated Meltdown on a Samsung Galaxy S7 mobile phone running LineageOS Android 14.1 with a Linux kernel 3.18.14. The device is equipped with a Samsung Exynos 8 Octa 8890 SoC consisting of a ARM Cortex-A53 CPU with four cores as well as an Exynos M1 “Mongoose” CPU with four cores [1]. While we were not able to mount the attack on the Cortex-A53 CPU, we successfully mounted Meltdown on Samsung's custom cores. Using exception suppression via branch misprediction as described in Section 4.1, we successfully leaked a pre-defined string using the direct physical map located at the virtual address `0xffff ffbf c000 0000`.

Containers. We evaluated Meltdown in containers sharing a kernel, including Docker, LXC, and OpenVZ and found that the attack can be mounted without any restrictions. Running Meltdown inside a container allows to leak information not only from the underlying kernel but also from all other containers running on the same physical host. The commonality of most container solutions is that every container uses the same kernel, *i.e.*, the kernel is shared among all containers. Thus, every container has a valid mapping of the entire physical memory through the direct-physical map of the shared kernel. Furthermore, Meltdown cannot be blocked in containers, as it uses only memory accesses. Especially with Intel TSX, only unprivileged instructions are executed without even trapping into the kernel. Thus, the confidentiality guarantee of containers sharing a kernel can be entirely broken using Meltdown. This is especially critical for cheaper hosting providers where users are not separated through fully virtualized machines, but only through containers. We verified that our attack works in such a setup, by successfully leaking memory contents from a container of a different user under our control.

6.2 Meltdown Performance

To evaluate the performance of Meltdown, we leaked known values from kernel memory. This allows us to not only determine how fast an attacker can leak memory, but also the error rate, *i.e.*, how many byte errors to expect. The race condition in Meltdown has a significant influence on the performance of the attack, however, the race condition can always be won. If the targeted data resides close to the core, e.g., in the L1 data cache, the race condition is won with a high probability. In this scenario, we achieved average reading rates of 552.4 KiB/s on average ($\sigma = 10.2$) with an error rate of 0.009 % ($\sigma = 0.014$) using exception suppression on the Core i7-8700K over 10 runs over 10 seconds. On the Core i7-6700K we achieved on average 515.5 KiB/s ($\sigma = 5.99$) with an error rate of 0.003 % on average ($\sigma = 0.001$) and 466.3 KiB/s on average ($\sigma = 16.75$) with an error rate of 11.59 % on average ($\sigma = 0.62$) on the Xeon E5-1630. However, with a slower version with an average reading speed of 137 KiB/s, we were able to reduce the error rate to zero. Furthermore, on the Intel Core i7-6700K if the data resides in the L3 data cache but not in the L1, the race condition can still be won often, but the average reading rate decreases to 12.4 KiB/s with an error rate as low as 0.02 % using exception suppression. However, if the data is uncached, winning the race condition is more difficult and, thus, we have observed average reading rates of less than 10 B/s on most systems. Nevertheless, there are two optimizations to improve the reading rate: First, by simultaneously letting other threads prefetch the memory locations [4] of and around the target value and access the target memory location (with exception suppression or handling). This increases the probability that the spying thread sees the secret data value in the right moment during the data race. Second, by triggering the hardware prefetcher within our own thread through speculative accesses to memory locations of and around the target value before the actual Meltdown attack. With these optimizations, we can improve the reading rate for uncached data to 3.2 KiB/s on average.

For all of the tests, we used Flush+Reload as a covert channel to leak the memory as described in Section 5, and Intel TSX to suppress the exception. For brevity, we omit the results of evaluating exception suppression using conditional branches. See Kocher et al. [14] for further information.

6.3 Limitations on ARM and AMD

We verified that some of the processors listed as not affected (see Table 1) are not vulnerable. Specifically, we experimented with some AMD and Arm-based processors, and were unable to reproduce Meltdown on those. We nevertheless note that for both ARM and AMD, the toy example as described in Section 3 works reliably, indicating that out-of-order execution generally occurs and instructions past illegal memory accesses are also performed.

6.4 Real-World Meltdown Exploit

To demonstrate the applicability of Meltdown, we show a possible real-world exploit that allows an attacker to steal the secret key used to store sensitive data. We look at VeraCrypt [9], a freeware solution that allows users to protect sensitive data using file or hard disk encryption. We note that VerCrypt is just an example, and any software that keeps its key material in main memory can be attacked in a similar manner.

Attack Scenario. In our scenario, the attacker gained access to the encrypted container or the encrypted hard drive of the victim. Without the secret key, the attacker is unable to decrypt the data, which is therefore secure. However, as VeraCrypt keeps the secret key in the main memory, relying on memory isolation to protect the key from unauthorized access, an attacker can use Meltdown to recover the key. A naive approach for that is to dump the entire physical memory of the computer and search in it. However, this approach is not practical. Instead, we show that the attacker can recover the page mapping of the VeraCrypt process and, thus, limit the amount of data to leak. For our experiments, we used VeraCrypt 1.22.

Breaking KASLR. As KASLR is active on the attacked system, the attacker first needs to de-randomize the kernel address space layout to access internal kernel structures and arbitrary physical addresses using the direct mapping. By locating a known value inside the kernel, e.g., the Linux banner, the randomization offset can be computed as the difference between the found address and the non-randomized base address. The Linux KASLR implementation only has an entropy of 6 bits [12], hence there are only 64 possible randomization offsets, making this approach practical.

Locating the VeraCrypt Process. Linux manages processes in a linked list whose head is stored in the `init_task` structure. The structure's address is at a fixed offset that only depends on the kernel build and does not change when packages are loaded. Each entry in the task list points to the next element, allowing easy traversal. Entries further include the process id of the task, its name and the root of the multi-level page table, allowing the attacker to identify the VeraCrypt process and to locate its page map.

Extracting the encryption key. The attacker can now traverse the paging structures and read the memory used by the process directly. VeraCrypt stores the `DataAreaKey` in a `SecureBuffer` in the `VolumeHeader` of a `Volume`. If ASLR is not active, the attacker can directly read the key from the offset where the key is located. Otherwise, the attacker searches the memory for a suitable pointer from which it can track the data structures to the stored key.

With the extracted key, the attacker can decrypt the container image, giving full access to the stored sensitive data. This attack does not only apply to VeraCrypt but to every software that keeps its key material stored in main memory.

7 Countermeasures

Fundamentally, Meltdown is a security issue rooted in hardware. Thus, to fully mitigate Meltdown, the hardware of modern CPUs needs to be modified. Indeed, since the original publication of Meltdown, Intel has released 9th generation i-cores, which contain hardware mechanisms that mitigate Meltdown.

For older vulnerable hardware, lower performing software mitigations do exist. More specifically, Gruss et al. [3] proposed KAISER, a kernel modification to not have the kernel mapped in the user space. While this modification was intended to prevent side-channel attacks breaking KASLR [4, 8, 12], it also prevents Meltdown, as it ensures that there is no valid mapping to kernel space or physical memory available in user space. Since the publication of Meltdown, Kernel Page Table Isolation (which is an implementation of KAISER) has been adopted by all major operating systems.

8 Conclusion

Meltdown fundamentally changes our perspective on the security of hardware optimizations that change the state of microarchitectural elements. Meltdown and Spectre teach us that functional correctness is insufficient for security analysis and the microarchitecture cannot be ignored. They further open a new field of research to investigate the extent to which performance optimizations change the microarchitectural state, how this state can be lifted into an architectural state, and how such attacks can be prevented. Without requiring any software vulnerability and independent of the operating system, Meltdown enables an adversary to read sensitive data of other processes, containers, virtual machines, or the kernel. KAISER is a reasonable short-term workaround to prevent large-scale exploitation of Meltdown until hardware fixes are deployed.

Acknowledgments

Several authors of this paper found Meltdown independently, ultimately leading to this collaboration. We want to thank everyone who helped us in making this collaboration possible, especially Intel who handled our responsible disclosure professionally, communicated a clear timeline and connected all involved researchers. We thank Mark Brand from Google Project Zero for contributing ideas and Peter Cordes and Henry Wong for valuable feedback. We would like to thank our anonymous reviewers for their valuable feedback. Furthermore, we would like to thank Intel, ARM, Qualcomm, and Microsoft for feedback on an early draft.

Daniel Gruss, Moritz Lipp, Stefan Mangard and Michael Schwarz were supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 681402).

Daniel Genkin was supported by NSF awards #1514261 and #1652259, financial assistance award 70NANB15H328 from the U.S. Department of Commerce, National Institute of Standards and Technology, the 2017-2018 Rothschild Postdoctoral Fellowship, and the Defense Advanced Research Project Agency (DARPA) under Contract #FA8650-16-C-7622.

References

[1] Burgess, B. Samsung Exynos M1 Processor. In *IEEE Hot Chips* (2016).

[2] Ge, Q., Yarom, Y., Cock, D., and Heiser, G. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *JCEN 8*, 1 (2018).

[3] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., and Mangard, S. KASLR is Dead: Long Live KASLR. In *International Symposium on Engineering Secure Software and Systems* (2017), Springer, pp. 161–176.

[4] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *CCS* (2016).

[5] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA* (2016).

[6] Gruss, D., Spreitzer, R., and Mangard, S. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium* (2015).

[7] Hennessy, J. L., and Patterson, D. A. *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. Morgan Kaufmann, San Francisco, CA, USA, 2011.

[8] Hund, R., Willems, C., and Holz, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *S&P* (2013).

[9] IDRIX. VeraCrypt, https://veracrypt.fr 2018.

[10] Intel. An introduction to the intel quickpath interconnect, Jan 2009.

[11] Intel. Rogue system register read, https://software.intel.com/security-software-guidance/software-guidance 2018.

[12] Jang, Y., Lee, S., and Kim, T. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *CCS* (2016).

[13] Kiriansky, V., and Waldspurger, C. Speculative buffer overflows: Attacks and defenses. CoRR arXiv 1807.03757, 2018.

[14] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre attacks: Exploiting speculative execution. In *S&P* (2019).

[15] Miller, M. Speculative store bypass, https://blogs.technet.microsoft.com/srd/2018/05/21/analysis-and-mitigation-of-speculative-store-bypass 2018.

[16] Osvik, D. A., Shamir, A., and Tromer, E. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA* (2006).

[17] Phoronix. Linux 4.12 To Enable KASLR By Default, https://www.phoronix.com/scan.php?page=news_item&px=KASLR-Default-Linux-4.12 2017.

[18] Schwarz, M., Schwarzl, M., Lipp, M., and Gruss, D. NetSpectre: Read arbitrary memory over network. CoRR arXiv 1807.10535, 2018.

[19] Sorin, D. J., Hill, M. D., and Wood, D. A. *A Primer on Memory Consistency and Cache Coherence.* 2011.

[20] Stecklina, J., and Prescher, T. LazyFP: Leaking FPU register state using microarchitectural side-channels. CoRR arXiv 1806.07480, 2018.

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

[22] Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T. F., and Yarom, Y. Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution, https://foreshadowattack.eu/foreshadow-NG.pdf 2018.

[23] Yarom, Y., and Falkner, K. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium* (2014).

[24] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *CCS* (2014).