Meltdown

CS305/341 Project

Team:

Guttu Sai Abhishek 180050036
Hrithik Vikas Samala 180050038
Mulinti Shaik Wajid 180050063
Sai Phanindra Ramasahayam 180050084
Sanapathi Sumanth Balaji 180050091

Overview:

1. The crux of the problem
2. Out-of-order Execution
3. Side Channel attacks
4. Flush + Reload attack
5. Meltdown
6. Countermeasures
7. Observations from paper
8. Appendix (PoC)

References:

Paper: https://meltdownattack.com/meltdown.pdf

Code: https://github.com/IAIK/meltdown/blob/master/README.md

https://en.wikipedia.org/wiki/Side-channel_attack

https://www.youtube.com/watch?v=UmLB1EWelCw&ab_channel=NPTEL-NOCIITM

The crux of the problem:

Today’s operating systems use memory isolation where each of the user processes has its address space which contains user code, data, stack, heap, kernel code and cannot access the address space of any other process. Also, a process has a privilege level, which is lower when it is in user mode and higher when the process is in kernel mode, and a context switch from one process to another can happen only when the process is in kernel mode. This is the central security feature where one process in user mode cannot access the memory of any other user process or the memory of the OS. The address space of kernel contains pointers to the entire physical memory, so to read memory of other processes it is sufficient if we can read memory of the kernel.

Recently in 2018, an attack called Meltdown was discovered. This attack can overcome the memory isolation provided by the operating systems and can read the entire kernel memory while being in a user process. So, it can be said that Meltdown is a bug at hardware level i.e., it doesn't exploit any software vulnerability so meltdown affects all operating systems, instead, Meltdown exploits the side-channel information available on modern processors. The root cause and the strength of this attack come from the side effects of the **out-of-order execution.**

Out-of-order execution is an important performance feature of today's processors in order to overcome the delays caused by busy execution units. For example, instead of waiting for an instruction which needs data to be fetched from main memory, the instructions next to it, if independent to the previous ones, are executed. But for some reason, if the CPU decides that the subsequent instructions it has executed are not actually needed/ should be avoided, then the CPU reverts back the architectural state of the process to ensure the correctness. But the microarchitectural state has some **residues** of the instructions executed and can be exploited to leak the data in kernel address space.

Out-of-order Execution:

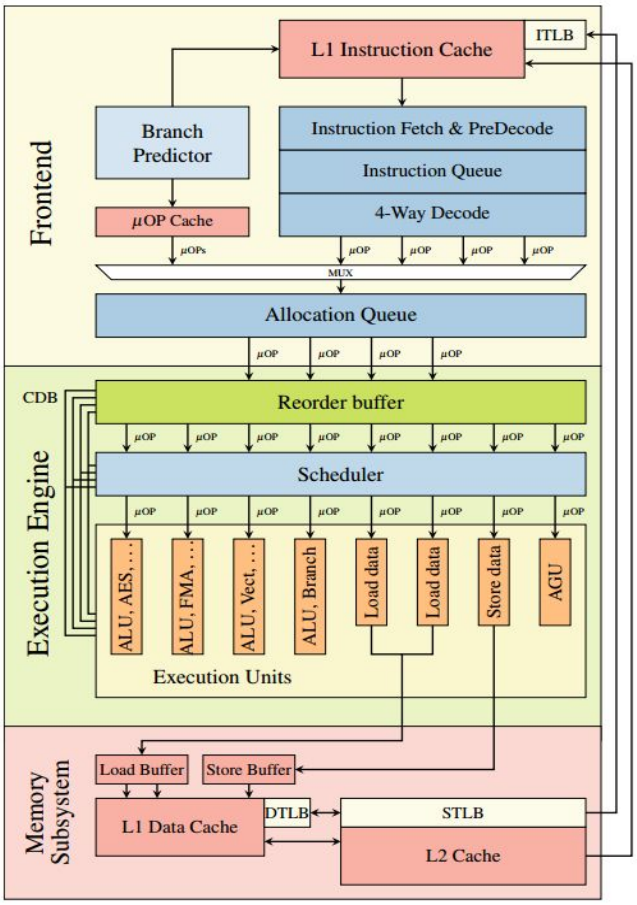
The figure shown below illustrates a simplified single core of Intel's Skylake microarchitecture.

Out-of-order execution is an optimization technique that allows maximizing the utilization of all execution units of a CPU core as exhaustive as possible.

x86 instructions are fetched in frontend and are decoded to micro-operations(μOPs). These are forwarded to the execution engine where the out-of-order execution takes place.

The **Reorder Buffer** is responsible for register allocation, register renaming and retiring and some optimisations. Then μOPs are forwarded to the Unified Reservation Station (**Scheduler**) that queues the operations on exit ports that are connected to Execution Units.

When some instruction is in need of access to data memory or is a branch instruction, the subsequent instructions are executed in parallel so as to avoid stalls. But if the branch prediction is incorrect or the next instructions aren't supposed to execute(maybe due to an interrupt in previous instruction) then the reorder buffer rollback to a sane state by clearing the reorder buffer and re-initializing the unified reservation station.



Side Channel attacks:

In computer security, a side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs). Timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information, which can be exploited.

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.

Flush + Reload attack:

It is a side-channel attack that targets a shared cache(L3 cache on a processor with 3 level caches). The attacker initially flushes out(using clflush) the address whose

presence/absence is to be detected in the cache, and then when the attackers try to access that memory address again(after some time), he/she notes the time of access.

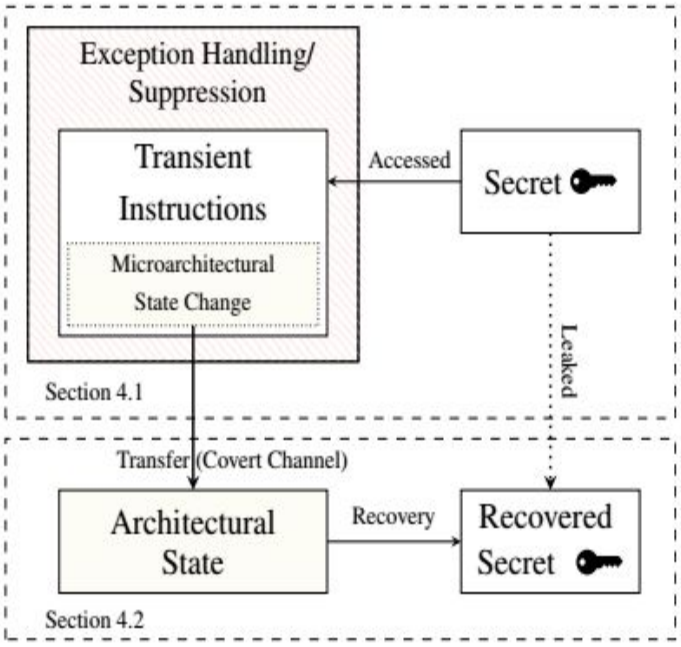
This access time would be less if that address is accessed by some process in the meantime(and therefore is present in the cache), through which he can conclude some information that he is not supposed to have access to. We are using this version of the side-channel attack in the subsequent pages about meltdown, as it allows us to build a fast and low-noise covert channel.

Meltdown:

Below are two principal building blocks in performing a meltdown attack.

1. The first is to perform a sequence of instructions, called transient instructions, which are executed out-of-order to leave a microarchitectural state change and cope up with the exception.

2. The second is to transfer the above information into an architectural state by building a covert channel.



When the user tries to access user-inaccessible kernel pages, an exception is triggered which leads to the termination of the process. Below are two ways to cope with this exception.

a. Exception handling:

A trivial way is to fork a child before performing invalid access and execute the instruction generating the exception and the transient instructions in this child. This child is then terminated when an exception is raised and the parent can recover the microarchitectural state. As L3 cache is shared the memory accessed by the child can also be read by the parent even if the child is executed on another CPU. Another way is

to include a signal handler that is executed when an exception occurs and can avoid crashing.

b. Exception suppression:

This approach prevents the exception from being raised in the first place. Transactional memory allows grouping memory accesses into one seemingly atomic operation, giving the option to roll-back to a previous state if an error occurs.

Covert channel:

Now we can have access to this information, which is in the form of microarchitectural state, by building a covert channel. The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel deduces the secret from the microarchitectural state change. If we consider the microarchitectural state as cache state then the transient instructions may perform array access, whose index depends on the secret value, in such a way that the index of access has an injective map to the basic unit of the cache.

For example, if the secret value is 8 bits then access can be of type probe_array[secret_value*page_size] i.e., for every 256 values of the secret value, a different page of main memory is accessed. By performing Flush+Reload on all of the 256 possible cache lines, the receiver can deduce the secret value. It is good to have addresses present on different pages to avoid false positives because of prefetcher

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.

Now that the building blocks of the attack are completed, we proceed to understand the attack in detail.

Attack Description:

Step 1: Reading the secret

In modern operating systems the entire kernel is mapped into the virtual address space of every user process and hence all kernel addresses lead to a valid physical address while translating them. The attackers targets to read this kernel information, between

the illegal memory access and raising of the exception, taking the advantage of out-of-order execution.

Let's say that, in this step, the attacker reads a kernel memory address which contains a value, say secret_value. This instruction is decoded into μOPs and is sent to reorder buffer. Due to out-of-order execution, the transient instructions are also decoded into μOPs and are queued in reservation station, waiting for its operand values that are not computed yet. We will see more about these transient instructions in step 2. Finally, when the exception occurs, all the results of the current instruction and transient instructions are flushed out and the exception may lead to termination or can be handled as described before.

Step 2: Transmitting the secret

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 contains indirect memory access to an address which is computed based on the secret value, so that a particular index of the probe array is accessed and is, therefore, placed in the cache.

More specifically, once the secret_value is successfully read from the step 1 the transient instruction, probe_array[secret_value*page_size], is ready to execute. The μOPs of the transient instructions are set out for execution and changes the cache state.

Also, note that the index is secret_value*page_size and not just secret_value. This is because when an address is cached, some of its neighbouring lines are also cached but this neighbouring line can’t come from a different page. So multiplying with page_size ensures that a different page(or part of it) is cached for different values of secret_value. Since step 3 takes a lot of time for a larger size of the secret value, it is better to access one bit of secret_value at a time by using that bit in indexing of the probe array.

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.

Step 3: Receiving the secret

The information in the cache can be accessed using Flush+Reload, as described in covert channel section, by iterating over all the possible indices of the probe array whose number depends on the size of secret value accessed. For example, if 5 bits of secret value is accessed at once, we need to check the access time for 32 different pages.

Bias towards 0 and optimisation:

From the paper, it is observed that there is an inherent bias towards the secret value being 0. The reason for this bias to '0' may either be that the memory load is masked out by a failed permission check, or a speculated value because the data of the stalled load is not available yet. So it is better to reread the secret value if we deduce that the secret value is zero from step 3. The maximum number of retries is an optimization parameter influencing the attack performance and the error rate.

This loop of retries is terminated either by reading a non-zero value or by the raised exception of the invalid memory access. In either case, the time until exception handling or exception suppression returns the control flow is independent of the loop after the invalid memory access, i.e., the loop does not slow down the attack measurably. Hence, these optimizations may increase attack performance.

Single bit transmission:

Since Flush+reload takes a lot of time for a large size of the secret value, it is better to transmit a single bit of the secret value. But transmitting a single bit makes it difficult to resolve whether the bit is an actual 0 or a bias towards 0 when the value found in step 3 is 0. Having a higher number of bits to transfer at a time reduces the chances that the actual combination is a decimal 0. This is a **trade-off** between implicit error-reduction and the overall transmission rate of the covert channel. However, from paper, it is observed that using single-bit transmission and some optimisations, like the one mentioned above, are giving low error rates.

Dealing with KASLR:

In 2013, kernel address space layout randomization (KASLR) was introduced to the Linux kernel allowing to randomize the location of kernel code at boot time. Now the attacker needs to find the offset before dumping the memory. However, the randomization is limited to 40 bit. Thus, if we assume a setup of the target machine with 8 GB of RAM, it is sufficient to test the address space for addresses in 8 GB steps. This

allows covering the search space of 40 bit with only 128 tests in the worst case. Hence KASLR is not effective in protecting against meltdown.

Countermeasures:

Disabling out-of-order execution or serialising the permission check and memory access can affect the performance of the modern CPUs and hence these are not simple trade-offs.

A more realistic solution would be to introduce a hard split of user space and kernel space. If the hard split bit is set, the kernel has to reside in the upper half of the address space, and the user space has to reside in the lower half of the address space. Then the privilege level can be directly derived from the virtual address without any further lookups.

A software patch called KAISER(Kernel Address Isolation to have Side-channels Efficiently Removed) can be used where the entire kernel is not mapped into the virtual address space of the user-mode but only a required part of it(a separate page table is maintained for kernel-mode where the entire kernel is mapped). Though this part doesn't contain any sensitive information they still might contain pointers which can leak information about randomised offsets and break KASLR.

A solution to this would require trampoline locations for every kernel pointer, i.e., the interrupt handler would not call into kernel code directly, but through a trampoline function. The trampoline function must only be mapped in the kernel. It must be randomized with a different offset than the remaining kernel. Consequently, an attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. Such trampoline code is required for every kernel memory that still has to be mapped in user space and contains kernel addresses. This approach is a **trade-off** between performance and security.

Observations from the paper:

Meltdown can’t be mounted on ARM or AMD cores. It can be assumed that the execution units for the load and the TLB are designed differently on ARM, AMD and Intel and, thus, the privileges for the load are checked differently and occurring faults are handled differently, e.g., issuing a load only after the permission bit in the page table entry has been checked.

Appendix

Proof Of Concept:

The git-hub repository provided with the paper demonstrates the proof of concept for the meltdown. In the repository, a new library is defined for this attack called libkdump.h which can be used to access information in any physical address. Specifically, the building blocks as defined in the paper which is “executing transient instructions” and “exception handling/suppression” are worked out over here. Here a struct called libkdump_config_t stores information such as measurement, retries, the physical offset. These quantities help in determining how many times and what information to accept for the address from the flush and reload.

The function libkdump_virt_to_phys takes a virtual address as an argument and returns the physical address or error code and requires root privileges to get this information. The function libkdump_phys_to_virt takes a physical address and returns a virtual address by adding the offset.

The main function defined in “libkdump.h” is the libkdump_read which is at the heart of this attack and it returns a character from the given virtual address. This function makes use of either the libkdump_read_signal_handler or libkdump_read_tsx depending on which is available but the core idea of traversing every page starting from the offset is implemented in these functions. Whether to use TSX or signal handler is determined by which is available. Transactional Synchronisation Extensions (TSX) is an extension to x86 ISA(Intel) and adds hardware transactional memory support, speeding up the execution of multi-threaded software through lock elision. Signal Handler is handling a certain exception and here if TSX is not available then the exception handling is set to signal handling. TSX is a variant of exception suppression while signal handling is exception handling.

Both the TSX and signal_handler use the flush and reload covert channel to gather information about the changes in microarchitectural state and return that to read function where after some retries to that address and if the same value is returned beyond a threshold(accept_after) then it is chosen as the value in that address.

Also, to demonstrate the working of meltdown attack we have:test.c, kaslr.c, reliability.c, physical_reader.c, memdump.c. First of all, KAISER (in Linux named KPTI) should be turned off and then we need to find the direct physical map offset using kaslr.c.

reliability.c helps us in determining whether the offset obtained is the direct physical map offset.

Some code snippets:

While executing transient instructions, to deal with exceptions there are two ways

1. Exception Handling
2. Exception Suppression

In the Exception Handling method, just before executing transient instructions the process is forked and the transient instructions are executed only in the child and after execution of transient instructions. We can also use signal handlers instead of doing fork which is a costly operation

Exception Suppression can be achieved in 1 way

1. Using Transactional memory(TSX)

We are using Signal handlers because we do not have access to TSX machines.
In the code signal handler is installed in the file libkdump.c at line number 419 as shown below, only SIGSEGV error is handled because by default when illegal memory is accessed SIGSEGV signal is sent to the process.

```
418     if (config.fault_handling == SIGNAL_HANDLER) {
419       if (signal(SIGSEGV, segfault_handler) == SIG_ERR) {
420         debug(ERROR, "Failed to setup signal handler\n");
421         libkdump_cleanup();
422         return -1;
423       }
424       debug(SUCCESS, "Successfully setup signal handler\n");
425     }
426     return 0;
427   }
```

The transient instructions which are executed in the attack are shown below

```
58  #define meltdown_nonull
59    asm volatile("1:\n"
60                 "movzx (%%rcx), %%rax\n"
61                 "shl $12, %%rax\n"
62                 "jz 1b\n"
63                 "movq (%%rbx,%%rax,1), %%rbx\n"
64                 :
65                 : "c"(phys), "b"(mem)
66                 : "rax");
```

The secret value is located at the address %rcx and in movzx in line 60 moves the secret value to register rax, ideally instructions after line 60 should not be executed because movzx instruction raises a trap and the process must move to kernel mode, but there is delay between moving the value and privilege checking and the cpu does not want to waste time until privilege checking is completed and it starts to execute instructions after movzx and these instructions are called transient instructions and the reason they g
et executed is because the CPU does “out of order execution”.
In line 61 the value at the secret address is left shifted by 12 places, this is because each page has 4kB size.

```
static int __attribute__((always_inline)) flush_reload(void *ptr) {
  uint64_t start = 0, end = 0;

  start = rdtsc();
  maccess(ptr);
  end = rdtsc();

  flush(ptr);

  if (end - start < config.cache_miss_threshold) {
    return 1;
  }
  return 0;
}
```

If the time stamp counter difference between end and start is less than cache_miss_threshold then it is concluded that the address is present in cache else it is not present in cache