Meltdown Explained

Meltdown Explained

This room explains the technical details behind the Meltdown vulnerability.

[Task 1] Introduction

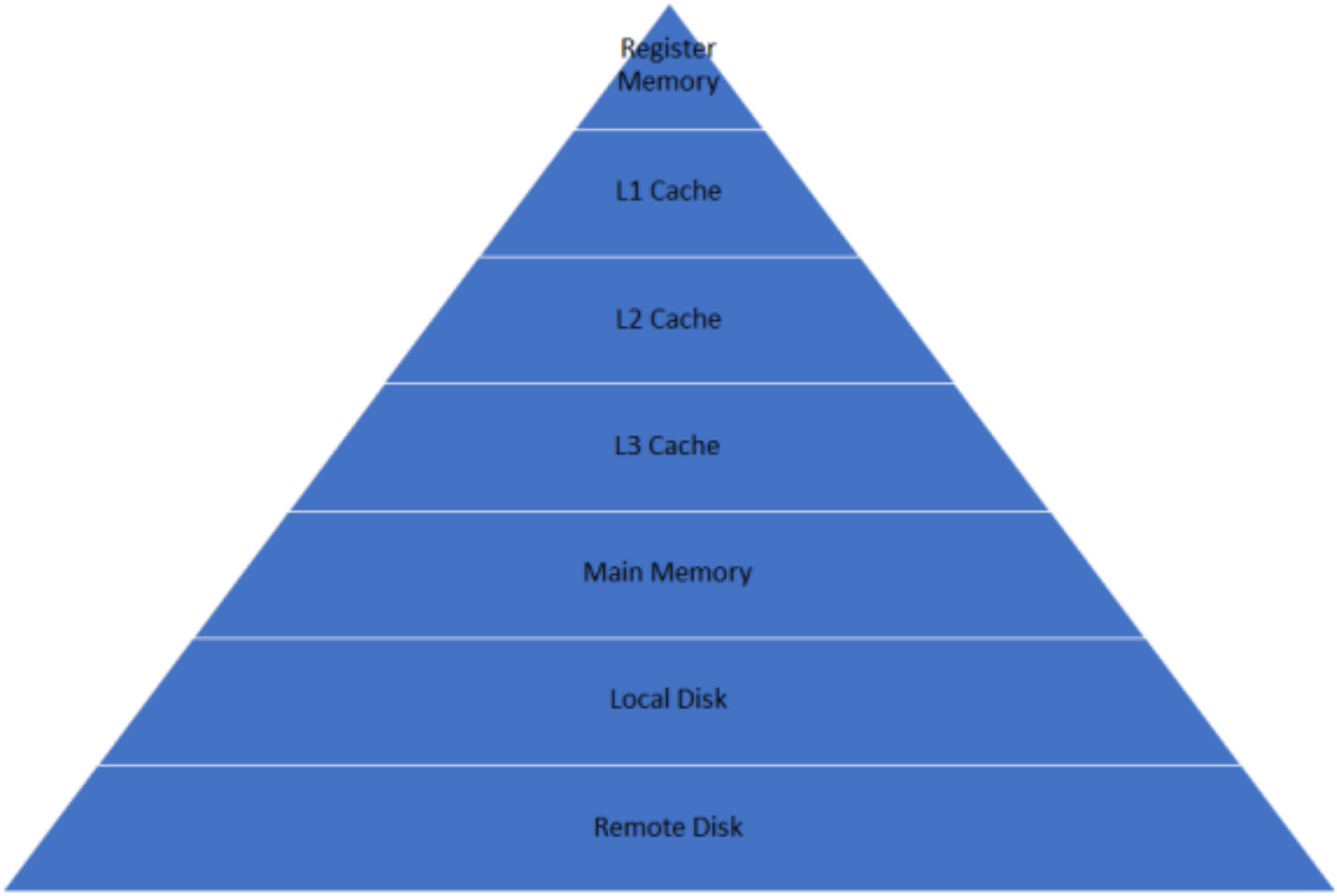
Meltdown is a vulnerability in CPU hardware that allows a malicious program to read kernel memory(as shown in this diagram). Kernel memory maps to all physical memory, which means that a malicious program could read memory from all processes'(which would not be possible otherwise). All Intel CPUs and some ARM CPUs are vulnerable to Meltdown, but various companies have release patches for this vulnerabilities.

#1
Read the above.

No answer needed

[Task 2] Background: CPU Caching

An oversimplification of how programs work are that a CPU would execute the programs instructions and these instructions would be stored in some sort of memory. In reality, memory is a hierarchy of different storage, each with their own access times and capacities.



The diagram above shows a generalised version of the memory hierarchy of a computer system. The registers would hold frequently accessed memory retrieved from the caches below which hold a subset of data retrieved from disk. The idea behind the memory hierarchy is that the storage devices get slower and cheaper as we move from the registers to the disks. Instead of the programming accessing disk or main memory constantly to retrieve some data or instructions, the storage levels at higher levels act as a cache. For example, the L1 memory acts as a cache for registers.
The memory at each level is divided into contiguous memory regions called blocks. Each block is uniquely referenced, which makes it easy to access. Like mentioned earlier, data at a higher level uses data at a lower level as a cache. If data at L2 memory needs to access data at main memory, if first checks if the data is at L3 memory(which acts as a cache because it is on a higher level). If the data exists in the L3 memory, then it is said to be a cache hit and it reads the memory from L3(which is faster from reading the memory at L2). If the data is not cached at L3 memory, then we have to fetch the data block from main memory and add it to L3(which may overwrite some memory on L3). This is called a cache miss.

#1
what is it called when a program accesses a cache and finds the correct value?

hit

#2
what is it called when a program accesses a cache and doesn't find the correct value?

miss

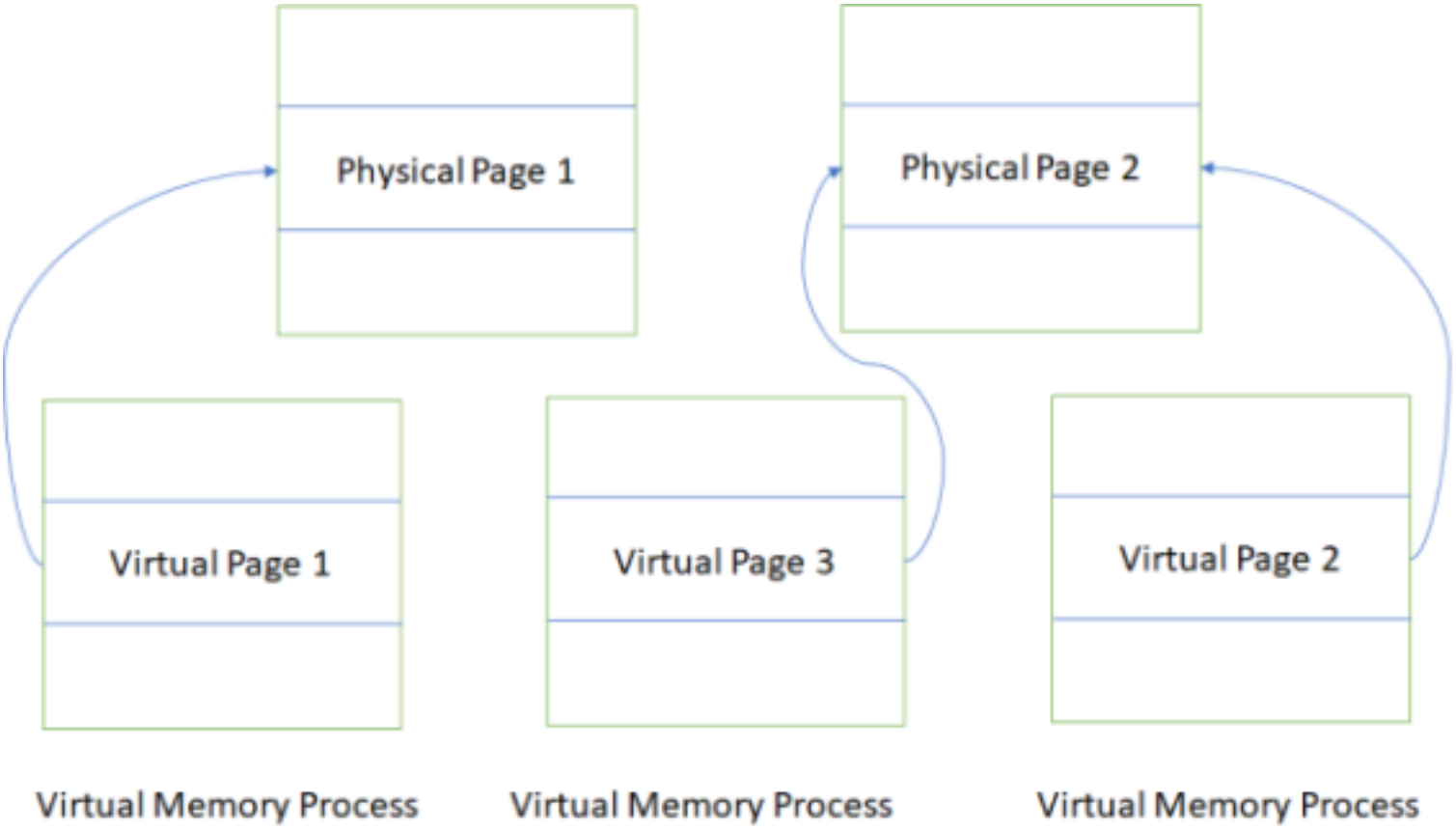
[Task 3] Background: Virtual Memory

Background: Virtual Memory

When a program runs, it does this in the context of a process. Each process has its own control flow(trace of execution depending on the instructions), and appears to have exclusive use of the main memory(this is provided by the kernel in the form of virtual memory). Since each process would need its own instructions and data(which is stored in memory), the kernel will load these instructions and data into a separate address space for each process and switch to this address space when the process is being executed. Using virtual memory means that:
• Each process gets the same uniform address space(which is just a set of contiguous bytes)

• Each processes virtual memory is isolated so once process cannot interfere with another's memory.

To actually implement this memory system, each processes virtual memory would have to map to the physical memory(where data and instructions are actually stored). This mapping is held by a data structure called a page table(which just contains page table entries that maps virtual pages to physical ages)



Like the diagram above shows, multiple virtual pages can map to the same physical page. The mapping of virtual from physical pages involves address translation to actually calculate the mapping to the physical address. Each page table entry also has permission bits:
• SUP bit: indicates whether the process must be running in kernel mode to access the page

• READ bit: indicates whether the process can read the page

• WRITE bit: indicates whether the process can write the bit

These permissions are checked when the address is translated from virtual to physical memory. The address translation is handled by what is called the Memory Management Unit(MMU).
It is important to note that with every virtual address space for every process, this virtual address space also contains kernel memory(that maps all physical RAM in the machine, so all processes memory). If a process in user mode tries to access the kernel memory, the memory management unit will throw an error(called a page fault) which will terminate the process. The reason kernel memory is included as part of a processes' virtual memory is effective performance; sometimes user programs need to carry out functions like reading from device memory or reading from a socket, which is done by entering the kernel.

#1
What kind of memory does the virtual address contain(apart from user memory)

kernel memory

[Task 4] Background: Pipelines & Out of Order Execution

When a CPU executes instructions, it tends to do it sequentially. While this sounds efficient, a single instruction may involve a different number of steps such as:
• Fetching the instruction

• Decoding the instruction

• Executing the instruction

• Writing result to memory

• Writing result to register

Each program is made up of a large number of instruction and because each instruction is sequentially executed, there may be inefficiencies within executing a single instruction; for example, the CPU may have to wait while fetching or decoding a particular instruction. This is where pipelining comes in:
• Divide the instruction execution steps(mentioned above) into a pipeline

• Execute these steps concurrently to ensure that there are no inefficiencies

• The shorter the stages, the more stages for different instruction can be completed at the same time, which leads to a faster instruction completion rate

Basic five-stage pipeline

Clock cycle / Instr. No.	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

To complement this approach of being more efficient, there may be points where entire instructions can halt, which cause the program to wait. To avoid the inefficiency while that particular instruction is still completing, the CPU can fetch instructions from later in their program and execute them now. In a situation where instructions produce data that may have to be written to memory, the CPU hardware ensures this is done in program order(i.e. It must wait for the previous instructions to complete). Additionally, if an instruction causes a hardware exception, for example when trying to access memory the program does not have access to, the instruction will still be executed by the results are squashed(which means they aren't written to registers, cache or memory).

#1
Read the above

No answer needed

[Task 5] Attack Explained

```
1  //some normal operations
2  a = b + c
3  d = g + f
4  x = a + y
5  if(x == 0){
6    i = kernel_memory[address]
7    j = i & 0x01
8    k = j * 4096
9    z = user_memory[k]
10 }
```

Consider the pseudo code above. Lines 2-4 can be considered normal arithmetic operations and the code in line 6-10 is executed based on the value of variable which is calculated at line 4. In the above section, we spoke about how a CPU may pipeline instructions to increase efficiency. Another way of increasing efficiency by reordering instructions is called speculative execution. In the example of the code above, while executing the other instructions, the CPU may speculate on the result of x. To avoid having to execute the instructions in the event that the condition is met, the CPU assumes that the condition is met and actually executes the instructions. In this case, line 6 involves reading from kernel memory, line 7 calculates and AND operation which either generates a 0 or a 1. Line 8 multiplies this by 4096 to ensure that the data is only fetched from one page(because the CPU can't fetch data across page boundaries). Line 9 then loads this into user memory and because the result from line 7 is either a 0 or a 1 so it will load memory from the location 0 or 4096. In normal cases, since the program is trying to access restricted memory, the hardware would throw a page fault and the process would terminate. Because the CPU is using speculative execution, it cannot confirm that the branch will actually be run, so it cannot really generate the fault.
The main aspect to focus on is that data in line 9 is loaded into user memory at either location 0 or 4096. From the background information, we know that it can take long to load from memory hence it is loaded into the cache to make access faster. At this point, the if statement terminates because it may not execute, or even if it does execute, it will still generate a page fault. However, there is a side effect here: the data is still loaded into the cache even though the instructions in the if statement may be executed. Either memory from location 0 or 4096 is in the cache and code can be written to check what location exactly is in the cache. This is determined using timing; the location that is in the cache will be accessed faster. This process can be used to read every bit from kernel memory at a time.

#1
Read the above

No answer needed

[Task 6] Mitigations

One of the major mitigations to prevent Meltdown is called Kernel Page Table Isolation(KPTI). KPTI involves separating kernel memory from being mapped to user space, and doing this means that user space can no longer access memory from the kernel. However, this comes with a performance cost - whenever the user program carries out operations that need access to the kernel e.g. making a system call, the user page has to switch to the kernel page and reverse this process when the operation is finished.

#1
Read the above

No answer needed

[Task 7] References

Meltdown and Spectre Video
Meltdown and Spectre Article
Computer Systems
KPTI Analysis
Proof of Concept

#1
Check out the other references

No answer needed