Indiana University Bloomington

CSCI-B544

Security for Networked Systems

---

# Spectre and Meltdown

---

Survey done by-
Ishneet Singh Arora
Nischith C N

Supervisor:
Dr Raquel Hill

# Contents


1 **Abstract** ..... 2

2 **Background** ..... 2
- 2.1 Virtual Memory ..... 2
- 2.2 CPU Cache ..... 4
- 2.3 Speculative Execution ..... 5
- 2.4 Side Channel Attacks ..... 6

3 **Introduction** ..... 7
- 3.1 Meltdown Vulnerability ..... 8
- 3.2 Spectre Vulnerability ..... 9
  - 3.2.1 Bounds Check Bypass(Spectre Variant 1) ..... 9
  - 3.2.2 Branch Target Injection(Spectre Variant 2) ..... 10

4 **Related Work** ..... 11
- 4.1 Meltdown Mitigtion ..... 11
- 4.2 Spectre Mitigation ..... 12
  - 4.2.1 Static Analysis and Fencing(Variant 1 Mitigation) ..... 12
  - 4.2.2 Retpoline(Variant 2 Mitigation) ..... 13
  - 4.2.3 IBRS,SITBP and IBPB(Variant 2 Mitigation) ..... 14

5 **Discussion** ..... 15
- 5.1 Spectre & Meltdown Checker ..... 15
- 5.2 Impact on Virtual & Cloud environments ..... 17

6 **Future Work** ..... 23

7 **Conclusion** ..... 23

8 **Reference** ..... 24

# 1. Abstract

In modern processors, Meltdown and Spectre exploit critical vulnerabilities. These hardware vulnerabilities allow programs to steal data processed from the computer. Although programs are usually not allowed to read data from other programs, a malicious program can use Meltdown and Spectre to store secrets in other running programs' memory. This may include your passwords, your personal photos, emails, instant messages and even business documents stored in a password manager or browser.

Meltdown and Spectrum work on personal, mobile and cloud computers. The meltdown breaks the most basic isolation between the user and operating system applications. This attack allows a program to access other programs and the operating system 's memory and thus also secrets. If your computer has a vulnerable processor and runs a non-patched operating system, it is not safe to use sensitive information without leaking the information. This applies to both personal and cloud-based computers. Fortunately, there are Meltdown software patches.

Spectre breaks the separation of applications. It enables an attacker to leak their secrets into error-free programs that follow best practices. Actually, the safety checks of these best practices actually increase the attack surface and can make applications more susceptible to Spectre. Spectre is harder to implement than Meltdown, but it is also harder to mitigate. However, specific known exploits based on Spectre can be prevented through software patches.

# 2. Background

## 2.1 Virtual Memory

Virtual memory is in existence since a very long time now and it has been used by all Operating Systems. It basically allows applications to utilize more memory than the machine actually has as it provides a layer of abstraction between memory address layout of what software’s see and the actual physical memory.

Figure 1: Virtual memory

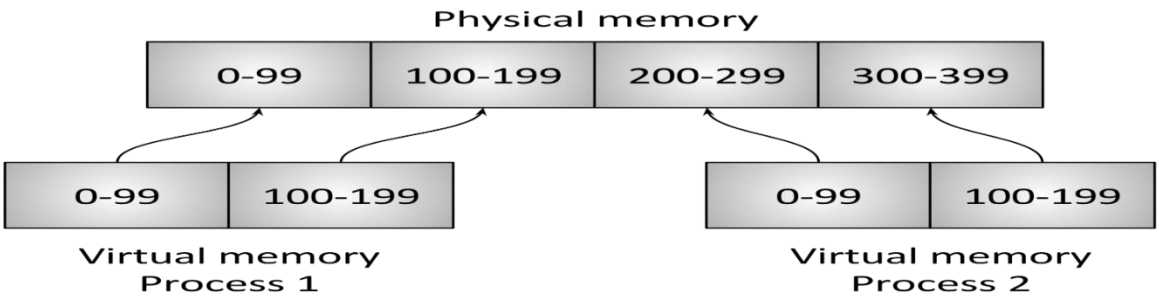
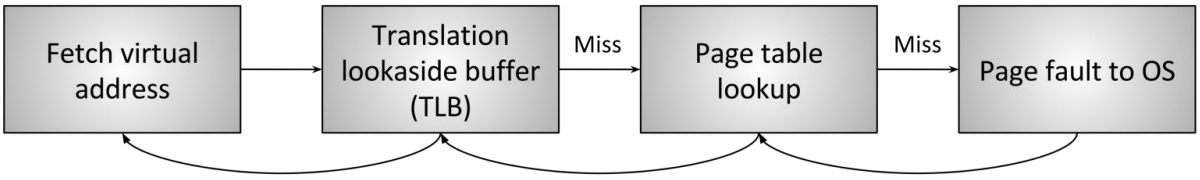



Figure 1 shows a simplistic computer with 400 bytes of memory laid out in “pages” of 100 bytes (real computers use powers of two, typically 4096). The computer has two processes, each with 200 bytes of memory across 2 pages each. The processes might be running the same code using fixed addresses in the 0–199-byte range, however, they are backed by discrete physical memory such that they don’t influence each other. Operating systems are abstracting the addresses that the application sees from the physical resources that back them.[1]

Modern CPU has a functionally where it maintains a Translation Lookaside Buffer (TLB) that caches recently used mappings. This is useful as it allows CPU to perform address translations directly.

Figure 2: Virtual memory translation




A program fetches the virtual address. CPU makes use of TLB to translate the address if found. If the address is not found, then checks with Page Tables to determine the mapping. If there isn't any mapping, then a "Page Fault" is raised to the OS which then takes a decision.

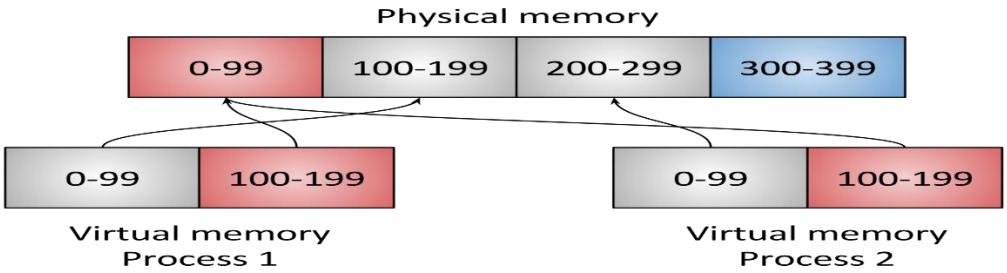


Figure 3: User/kernel virtual memory mappings

- Kernel memory is shown in red. Kernel memory is special memory that only the operating system should be able to access. User programs should not be able to access it.
- User memory is shown in gray.
- Unallocated physical memory is shown in blue.

In this example, we start seeing some of the useful features of virtual memory. Primarily:

- User memory in each process is in the virtual range 0–99 but backed by different physical memory.
- Kernel memory in each process is in the virtual range 100–199 but backed by the *same* physical memory.

Even though kernel memory is mapped into each user process, it cannot access the kernel memory when the process is running in user mode. If a process attempts to do so, it will trigger a page fault and the operating system will terminate it. However, when the process is running in kernel mode (for example during a system call), the processor will allow the access.

## 2.2 CPU Cache

In the past few years optimization of the processors has been done drastically. However, the memory access and all the layers of storage is yet slower even though the clock speed has increased. Caching has been used to improve the performance. Caching does a lot to relieve this performance gap by storing data at each layer in a small amount of more expensive, faster storage on the CPU. It makes use of the concept of prefetching , where data that has recently been used or which might be used soon gets stored. Both Meltdown and Spectre exploit caching by timing how long it takes to access data from memory so one can tell if it came from RAM or the cache.

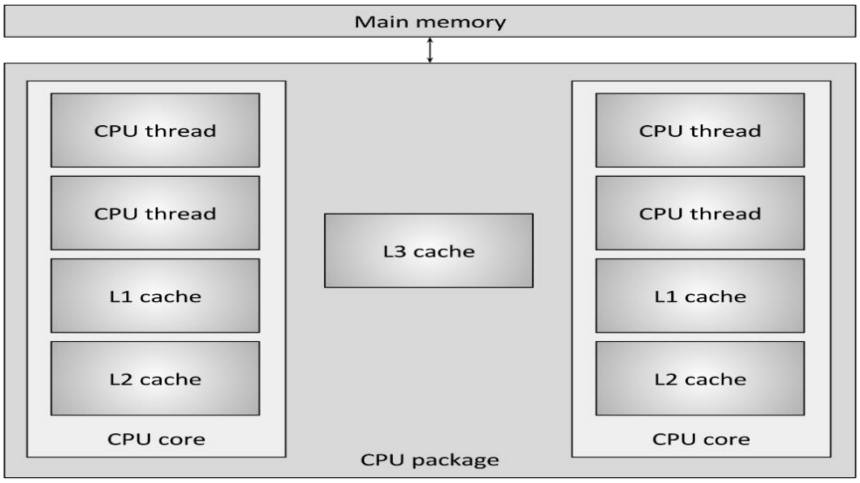


Figure 4: CPU thread, core, package, and cache topology.

Figure 4shows a generic topology that is common to most modern CPUs. It is composed of the following components:

- Each CPU core has 2 CPU thread per core which is the basic unit of execution.
- The cache levels closer to the CPU thread are smaller, faster, and more expensive. The further away from the CPU and closer to main memory the cache is the larger, slower, and less expensive it is.
- Each CPU thread on the core makes use of the same caches.
- All of the CPU cores in the package typically share an L3 cache

## 2.3 Speculative execution

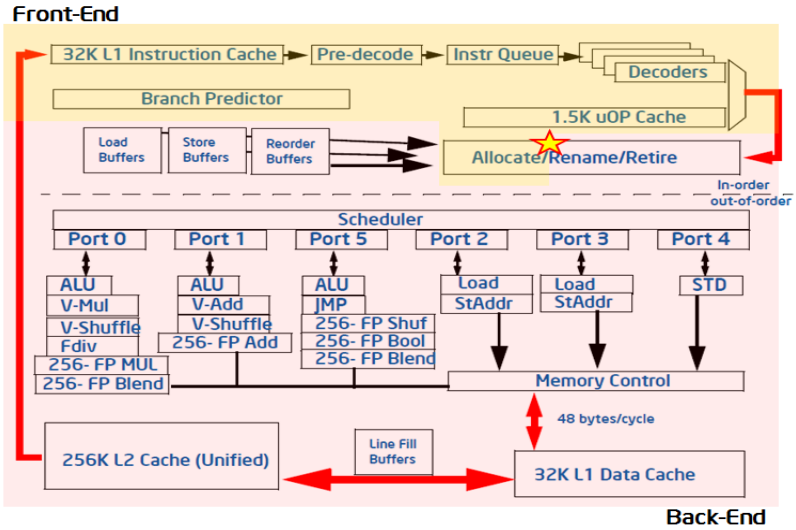



Figure 5: Modern CPU execution engine (Source: Google images)

The modern CPUs are incredibly complicated and do not simply execute machine instructions in order. Each CPU thread has a complicated pipelining engine that is capable of executing instructions out of order. The reason for this has to do with caching. As discussed in the previous section, each CPU makes use of multiple levels of caching. Each cache miss adds a substantial amount of delay time to program execution. In order to mitigate this, processors are capable of executing ahead and out of order while waiting for memory loads. This is known as speculative execution.[1]

```
If(x<array1_size){
y=array2[array1[x]*256];
 }
```

Consider the array1_size is not available in the cache but the array1 address is. Now CPU might speculate that x is less than array1_size and go ahead with the code and perform the conditions inside if statement.  Now when actually array1_size is read from memory and if the speculation

is correct then, in that case, we can move forward else we can discard this speculative calculation and start over. This will be like as if we had to wait for the value of array1_size from memory.

Another type of speculative execution is known as indirect branch prediction. This is extremely common in modern programs due to virtual dispatch.

```
class Base {
public:
virtual void Foo() = 0;
};
class Derived : public Base {
public:
void Foo() override { … }
};
Base* obj = new Derived;
obj->Foo();
```

Because this operation is so common, modern CPUs have various internal caches and will often guess where the indirect branch will go and continue execution at that point. Again, if the CPU guesses correctly, it can continue having saved a bunch of time. If it didn't, it can throw away the speculative calculations and start over.[1]

Difference between Out-Of-Order(OOO) and Speculative Execution

OOO execution is an execution model in which instructions can be executed in an order that is potentially different from the program order. However, the instructions are still retired in program order so that the program's observed behavior is the same as the one intuitively expected by the programmer. Speculative execution is an execution model in which instructions can enter the pipeline and even begin execution without even knowing for sure that they will indeed be required to execute.

## 2.4 Side Channel Attacks

Rather than weaknesses in the algorithm itself (e.g. cryptanalysis and software bugs) a side-channel attack is an attack based on information gained from the computer system. Timing information, power consumption, electromagnetic leaks or even sound can provide an additional information source that can be used.

Few parameters on which side channel can be based on-

- Cache attack — attacks based on attacker's ability to monitor cache accesses made by the victim in a shared physical system as in virtualized environment or a type of cloud service.
- Timing attack — attacks based on measuring how much time various computations (such as, say, comparing an attacker's given password with the victim's unknown one) take to perform.
- Power-monitoring attack — attacks that make use of varying power consumption by the hardware during computation.
- Acoustic cryptanalysis — attacks that exploit sound produced during a computation (rather like power analysis).
- Differential fault analysis — in which secrets are discovered by introducing faults in a computation.
- Optical - in which secrets and sensitive data can be read by visual recording using a high-resolution camera, or other devices that have such capabilities (see examples below).

# 3. Introduction

How the Spectre and Meltdown attacks use caching and speculation?

Theoretically accessing restricted memory during speculative execution is not a problem because either the code is not reached by the true execution path and the results are discarded or the results of any subsequent calculation are raised and discarded. Spectre and Meltdown are known as side-channel attacks because they exploit a vulnerability of a secure system. Normally a fault occurs when a restricted memory is accessed, and the execution is stopped. Once the fault is raised the results of speculative execution will be discarded, but the changes appended in the cache won't get reverted. One can then use the time taken to access the restricted memory to determine the cache. From here you can deduce something about the restricted data that was read during the speculative execution. This allows data to be read out bit by bit from forbidden areas of memory that could contain passwords and other sensitive operating system data.

What is the difference between Meltdown and Spectre?

Meltdown works by running an attack program that reads privileged data from the kernel address space–data that should only be accessible to the operating system. Spectre works by enabling a victim program to read data that would normally not access and leak data to a second attack program running on the same machine. Meltdown is simpler to implement. Spectre is CPU specific, much harder to implement

By exploiting Meltdown, an attacker can use a program running on a machine to gain access to data from all over that machine that the program shouldn't normally be able to see, including data belonging to other programs and data that only administrators should have access to. Meltdown doesn't require too much knowledge of how the program the attacker hijacks works, but it only works with specific kinds of Intel chips. This is a pretty severe problem, but fixes are being rolled out.[2]

By exploiting the Spectre variants, an attacker can make a program reveal some of its own data that should have been kept secret. It requires more intimate knowledge of the victim program's inner workings, and doesn't allow access to other programs' data, but will also work on just about any computer chip out there. Specter's name comes from speculative execution but also derives from the fact that it will be much trickier to stop — while patches are starting to become available, other attacks in the same family will no doubt be discovered. That's the other reason for the name: Spectre will be haunting us for some time.[2]

Why are Spectre and Meltdown dangerous?

Spectre and Meltdown both can cause dangerous attacks. For instance, JavaScript code on a website could use Spectre to trick a web browser into revealing user and password information. Attackers could exploit Meltdown to view data owned by other users and even other virtual servers hosted on the same hardware, which is potentially disastrous for cloud computing hosts. But beyond the potential specific attacks themselves lies the fact that the flaws are fundamental to the hardware platforms running beneath the software we use every day. Even code that is formally secure turns out to be vulnerable because the assumptions underlying the security processes have turned out to be false.

## 3.1 Meltdown Vulnerability

The first vulnerability, known as Meltdown, is surprisingly simple to explain and almost trivial to exploit. The exploit code roughly looks like the following[1]:

```
1. uint8_t* probe_array = new uint8_t[256 * 4096];
2. // ... Make sure probe_array is not cached
3. uint8_t kernel_memory = *(uint8_t*)(kernel_address);
4. uint64_t final_kernel_memory = kernel_memory * 4096;
5. uint8_t dummy = probe_array[final_kernel_memory];
6. // ... catch page fault
7. // ... determine which of 256 slots in probe_array is cached
```

Let's take each step above, describe what it does, and how it leads to being able to read the memory of the entire computer from a user program.

1. Here a “probe array” is allocated. This is the memory which will be used for a side channel to retrieve data from the kernel.

2. Following the allocation, the attacker makes sure that none of the memory in the probe array is cached. One could use CPU-specific instructions to clear a memory location from the cache.

3. The attacker then proceeds to read a byte from the kernel’s address space. Any such access will result in a page fault. That is indeed what will eventually happen at step 3.

4. However, modern processors also perform speculative execution and will execute ahead of the faulting instruction. Thus, steps 3–5 may execute in the CPU’s pipeline before the fault is raised. In this step, the byte of kernel memory (which ranges from 0–255) is multiplied by the page size of the system, which is typically 4096.

5. In this step, the multiplied byte of kernel memory is then used to read from the probe array into a dummy value. The multiplication of the byte by 4096 is to avoid a CPU feature called the “prefetcher” from reading more data than we want into the cache.

6. By this step, the CPU has realized its mistake and rolled back to step 3. However, the results of the speculated instructions are still visible in the cache.

7. In step 7, the attacker iterates through and sees how long it takes to read each of the 256 possible bytes in the probe array that could have been indexed by the kernel memory. The CPU will have loaded one of the locations into the cache and this location will load substantially faster than all the other locations (which need to be read from main memory). This location is the value of the byte in kernel memory.

## 3.2 Spectre vulnerability

Spectre shares some properties of Meltdown and is composed of two variants. Unlike Meltdown, Spectre is substantially harder to exploit, but affects almost all modern processors produced in the last twenty years. Essentially, Spectre is an attack against modern CPU and operating system design versus a specific security vulnerability.

### 3.2.1 Bounds check bypass (Spectre variant 1)

Bounds check bypass takes advantage of the speculative execution used in processors to achieve high performance. To avoid the processor having to wait for data to arrive from memory, or for previous operations to finish, the processor may speculate as to what will be executed. If it is incorrect, the processor will discard the wrong values and then go back and redo the computation with the correct values. At the program level this speculation is invisible, but because instructions were speculatively executed they might leave hints that a malicious actor can measure, such as

which memory locations have been brought into cache. Using the bounds check bypass method, malicious actors can use code gadgets ("confused deputy" code) to infer data values that have been used in speculative operations. This presents a method to access data in the system cache that the malicious actor should not otherwise be able to read. The bounds check bypass store variant makes an additional range of vulnerabilities possible by targeting variables on the stack, function pointers, or return addresses. This allows malicious actors to influence variables used later in speculative execution or to direct speculative execution to other areas of code, where malicious actors could then observe system behavior.

The first Spectre variant is known as “bounds check bypass.” This is demonstrated in the following code snippet (which is the same code snippet I used to introduce speculative execution above).

```
if (x < array1_size) {
 y = array2[array1[x] * 256];
 }
```

In the previous example, assume the following sequence of events:

1. The attacker controls x.
2. array1_size is not cached.
3. array1 is cached.
4. The CPU guesses that x is less than array1_size.
5. The CPU executes the body of the if statement while it is waiting for array1_size to load, affecting the cache in a similar manner to Meltdown.
6. The attacker can then determine the actual value of array1[x] via one of various methods.

Spectre is considerably more difficult to exploit than Meltdown because this vulnerability does not depend on privilege escalation. The attacker must convince the kernel to run code and speculate incorrectly. Typically, the attacker must poison the speculation engine and fool it into guessing incorrectly. That said, researchers have shown several proof-of-concept exploits.

### 3.2.2 Branch target injection (Spectre variant 2)

Branch Target Injection uses the microprocessor 's speculative execution behavior to expose some code to more information than intended. This method influences the microprocessor 's indirect branch predictor to execute speculative malicious code that leaves behind a microarchitectural state that the attacker can then use to infer data values.

A conditional direct branch only has two possible paths that can be speculatively executed. Unlike direct branches, an indirect branch can cause the microprocessor to speculatively execute a very wide range of possible targets. This attack is done by causing a direct branch to speculatively execute a segment of code. If the attacker carefully chooses the code that effectively result in Spectre Variant 1, then the attacker can infer sensitive data from the victim's memory space.

Indirect branching is very common in modern programs. Variant 2 of Spectre utilizes indirect branch prediction to poison the CPU into speculatively executing into a memory location that it never would have otherwise executed. If executing those instructions can leave state behind in the cache that can be detected using cache inference attacks, the attacker can then dump all of the kernel memory. Like Spectre variant 1, Spectre variant 2 is much harder to exploit than Meltdown, however, researchers have demonstrated working proof-of-concept exploits of variant 2.

# 4. Related Work

## 4.1 Meltdown Mitigations

Meltdown is easy to understand, trivial to exploit, and fortunately also has a relatively straightforward mitigation (at least conceptually — kernel developers might not agree that it is straightforward to implement).

**Kernel page table isolation (KPTI)**

Operating Systems use techniques where every kernel memory is mapped into user mode address space. This is done for high performance and for simplicity. By doing this when a program makes a System call, OS or the kernel doesn’t needs to perform any additional work. To prevent Meltdown this dual mapping can no longer be done.

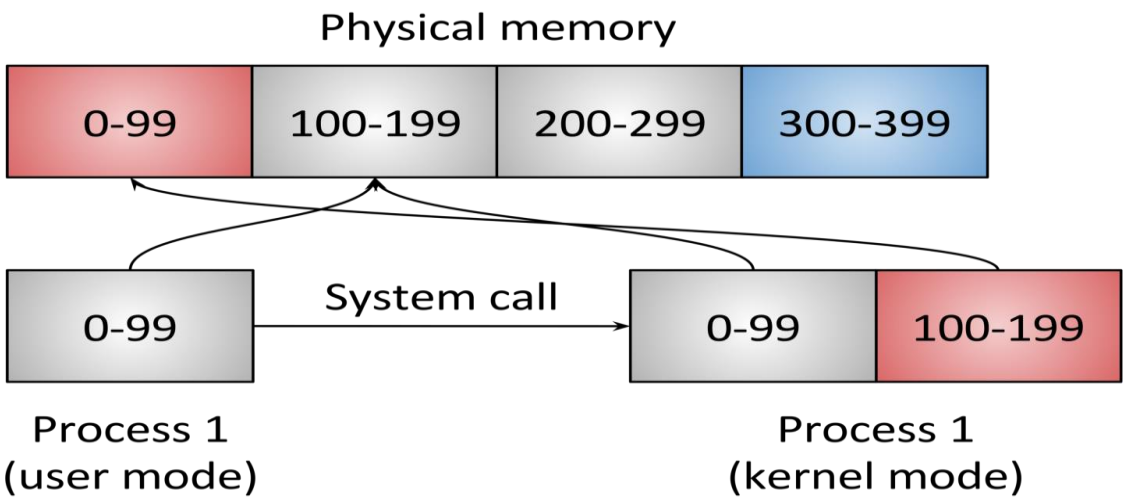



Figure 6: Kernel page table isolation

Figure 6 shows a technique called Kernel Page Table Isolation (KPTI). Here we no longer map kernel memory into a program when it is running in userspace. If there is no mapping present, speculative execution is no longer possible and will immediately result in a fault.

This modification considerably slows down the user mode to kernel mode transitions and vice versa due to the fact that now TLB needs to be flushed for each transition and also it needs to be modified.

The result of mitigation being the performance will be slower until future hardware is explicitly designed for the address space separation described.

Newer x86 CPUs have a feature is known as ASID (address space ID) or PCID (process context ID) that can be used to make this task substantially cheaper PCID allows an ID to be associated with a TLB entry and then to only flush TLB entries with that ID. The use of PCID makes KPTI cheaper, but still not free.

## 4.2 Spectre mitigations

The Spectre mitigations is more difficult to implement when compared to Meltdown mitigation. Intel and the major OS and cloud vendors have been working furiously for months to develop mitigations.

### 4.2.1Static analysis and fencing (variant 1 mitigation)

The only known variant 1 mitigation (bound bypass check) is a static code analysis to determine code sequences that could be controlled by an attacker to interfere with speculation. Vulnerable code sequences can have a serialization instruction like lfence inserted that stops the speculative

execution until all instructions have been executed up to the lfence. When inserting fence instructions, care must be taken, as too many can have serious effects on the performance.

### 4.2.2Retpoline (variant 2 mitigation)

The first Spectre variant 2 (branch target injection) mitigation was developed by Google and is known as “retpoline”. It was experimentally developed by Google and then verified by Intel hardware engineers.

Retpoline relies on the fact the calling and returning from functions and the associated stack manipulations are so common in computer programs that CPUs are heavily optimized for performing them. In a nutshell, when a “call” is performed, the return address is pushed onto the stack. “ret” pops the return address off and continues execution. Speculative execution hardware will remember the pushed return address and speculatively continue execution at that point.

The retpoline construction replaces an indirect jump to the memory location stored in register r11:

```
jmp *%r11
```

with:

```
1.call set_up_target;

4.capture_spec:

   pause;

   jmp capture_spec;

   set_up_target:

 2. mov %r11, (%rsp);

 3.ret;
```

Let’s see what the previous assembly code does one step at a time and how it mitigates branch target injection[1].

1. In this step the code calls a memory location that is known at compile time so is a hard coded offset and not indirect. This places the return address of capture_spec on the stack.

2. The return address from the call is overwritten with the actual jump target.

3. A return is performed on the real target.

4. When the CPU speculatively executes, it will return into an infinite loop! Remember that the CPU will speculate ahead until memory loads are complete. In this case, the speculation has been manipulated to be captured into an infinite loop that has no side effects that are observable to an attacker. When the CPU eventually executes the real return it will abort the speculative execution which had no effect.[1]

This mitigation is that it requires all software to be recompiled such that indirect branches are converted to retpoline branches. However for cloud services such as Google that own the entire stack, recompilation is not a big deal.

### 4.2.3IBRS, STIBP, and IBPB (variant 2 mitigation)

Intel has provided solutions by providing modifications to mitigate branch target injection attacks.

The following three new hardware features are included in the CPU microcode update:

- Indirect Branch Restricted Speculation (IBRS)
- Single Thread Indirect Branch Predictors (STIBP)
- Indirect Branch Predictor Barrier (IBPB)

What we could understand is-

IBRS clean the branch prediction cache between privilege levels( user to the kernel) and disable the sibling CPU thread branch prediction. Remember that every CPU core usually contains two CPU threads. The branch prediction hardware is shared between the threads on modern CPUs. IBRS in kernel mode essentially prevents any previous execution in user mode and an execution on the sibling CPU thread from affecting the prediction of the branch.

STIBP has additional functionality of IBRS that just disables branch prediction on the sibling CPU thread. This is useful to prevent a sibling CPU thread from poisoning the branch predictor when running two different user-mode processes (or virtual machines) on the same CPU core at the same time.

IBPB appears to flush the branch prediction cache for code running at the same privilege level. This can be used when switching between two user mode programs or two virtual machines to ensure that the previous code does not interfere with the code that is about to run.

One good solution is to have an IBRS “always on” model where the hardware just defaults to clean branch predictor separation between CPU threads and correctly flushes state on privilege level changes. Approximately 20% slowdown on certain system call heavy workloads was observed when the mitigations were rolled out.

# 5. Discussion

## 5.1 Spectre & Meltdown Checker

We have tried to explore further by searching for a way to learn if whether Our local desktops are still affected by Spectre and Meltdown. We have found a script which provide an in-depth analysis of your system to find out if the system is affected or not.

The script provides checks against below mentioned Spectre and Meltdown attacks and their variations-

- CVE-2017-5753 [bounds check bypass] aka 'Spectre Variant 1'
- CVE-2017-5715 [branch target injection] aka 'Spectre Variant 2'
- CVE-2017-5754 [rogue data cache load] aka 'Meltdown'
- CVE-2018-3640 [rogue system register read]
- CVE-2018-3639 [speculative store bypass]
- CVE-2018-3615 [L1 terminal fault] aka 'Foreshadow (SGX)'
- CVE-2018-3620 [L1 terminal fault] aka 'Foreshadow-NG (OS)'
- CVE-2018-3646 [L1 terminal fault] aka 'Foreshadow-NG (VMM)'

Steps we performed

- curl -L https://meltdown.ovh -o spectre-meltdown-checker.sh
- wget https://meltdown.ovh -O spectre-meltdown-checker.sh
- sudo ./spectre-meltdown-checker.sh

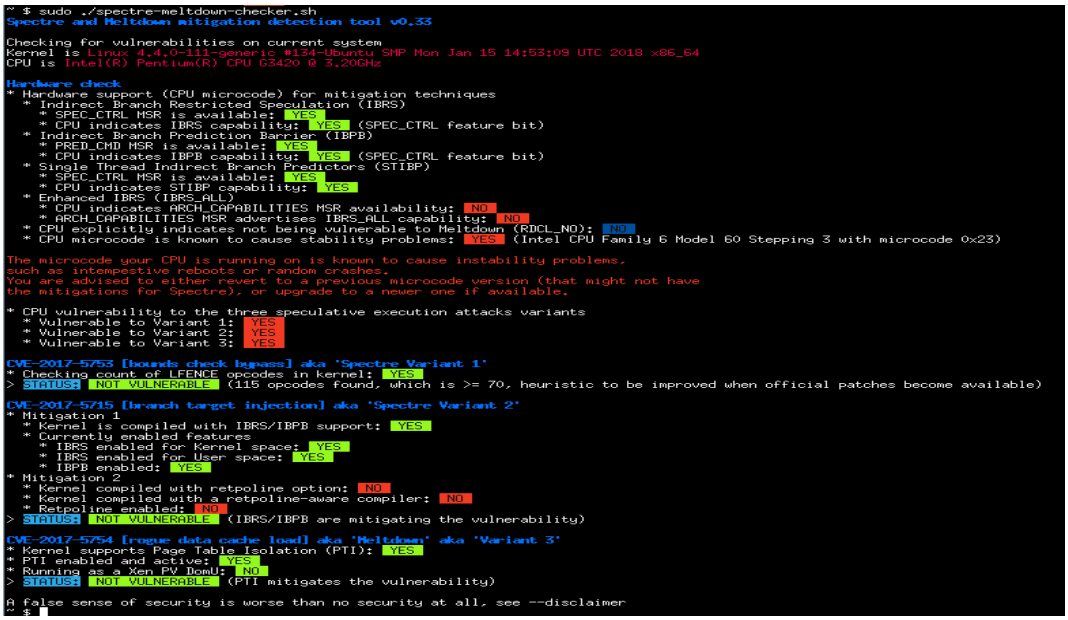
a)Run the script with docker container

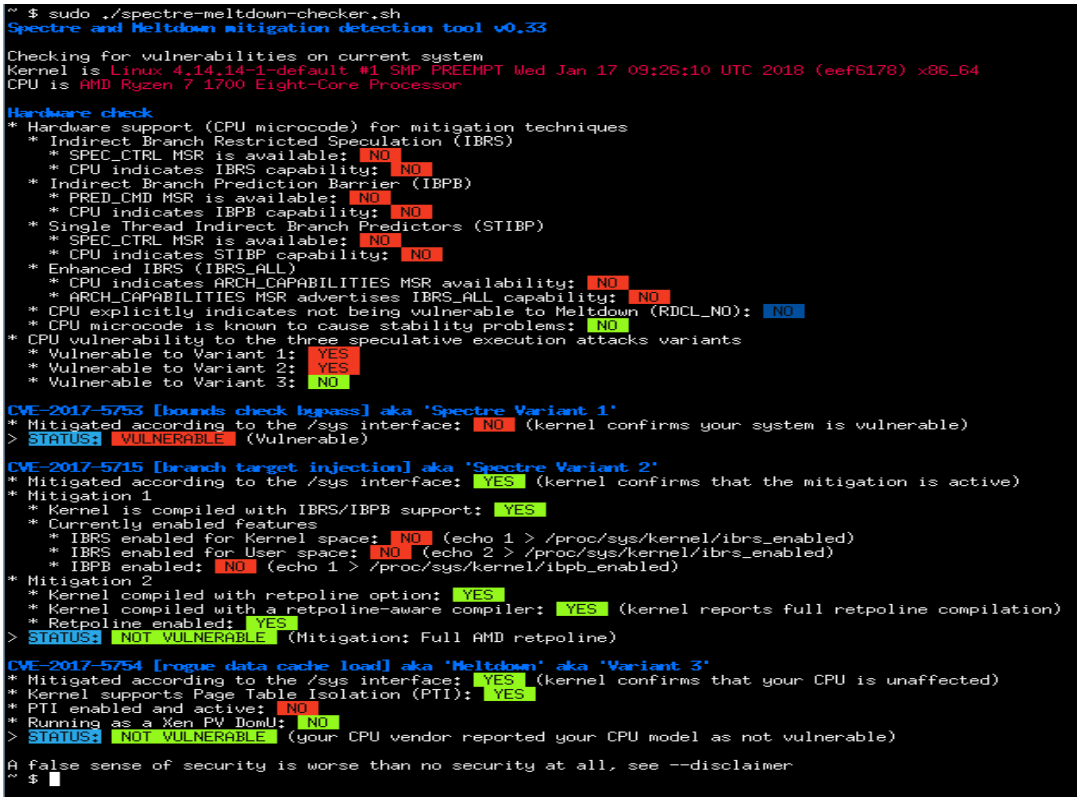
- docker-compose build

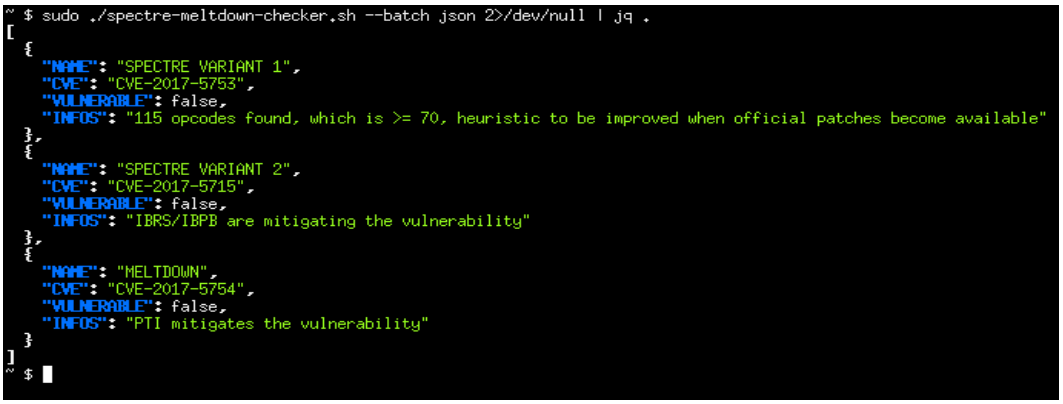
- docker-compose run --rm spectre-meltdown-checker

b)Run the script without docker-compose

- docker build -t spectre-meltdown-checker .
- docker run --rm --privileged -v /boot:/boot:ro -v /dev/cpu:/dev/cpu:ro -v /lib/modules:/lib/modules:ro spectre-meltdown-checkerSample output:






## 5.2 Impact on Virtual and Cloud environment and mitigation technique

Virtual machines, virtual appliances, hypervisors, server firmware, and CPU microcode were impacted. Cloud has various layer and components integrated together Spectre and meltdown had impact on both layer of virtualization i.e. ESXi hypervisor and guest operating system.

Configuration parameters used for testing the performance on various environments were as below:

- VMware ESXi Server:

- Supermicro SYS5208D-TN4T
- Intel(R) Xeon(R) CPU D-1541 @ 2.10GHz
- ESXi 6.5 Update 1 – version 5969303 (beginning test)
- Samsung 960 1 TB NVMe backed datastore
- No other VMs running

- VMware Virtual Machine:
  - Windows 10 Pro – 1709 (OS Build 16299.125) – beginning test
  - (4) vCPUs assigned, 8 GB of RAM
  - single 50 GB hard drive
  - Using Paravirtual Storage controller
  - Network adapter (VMXNET3)
  - VMware Compatibility version ESXi 6.5 (VM version 13)

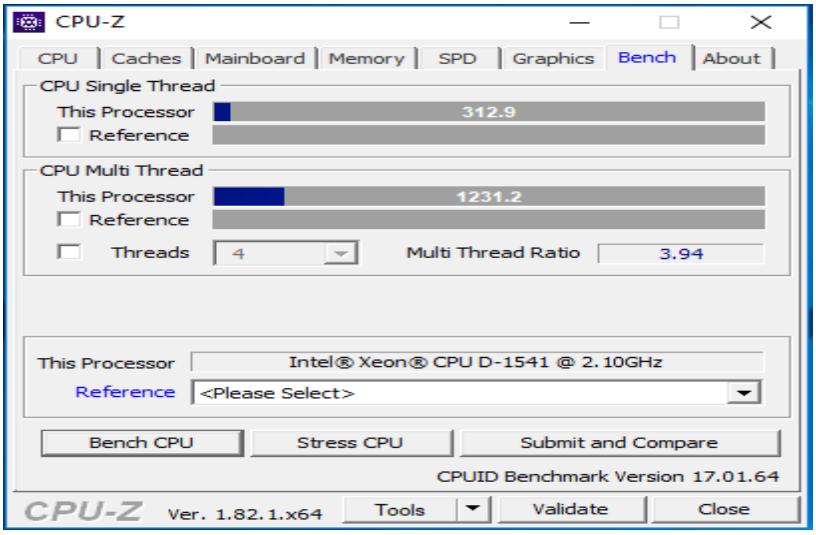
Tools they used for testing

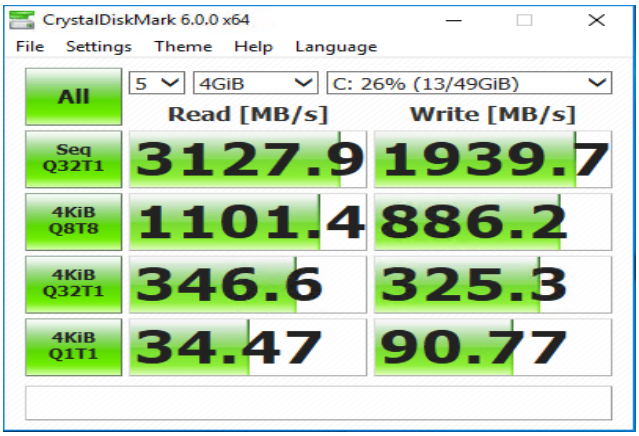
1. CPU-Z for a quick CPU benchmark

2. Crystal Disk Benchmark for disk benchmark statistics

**Screenshots before Patching**

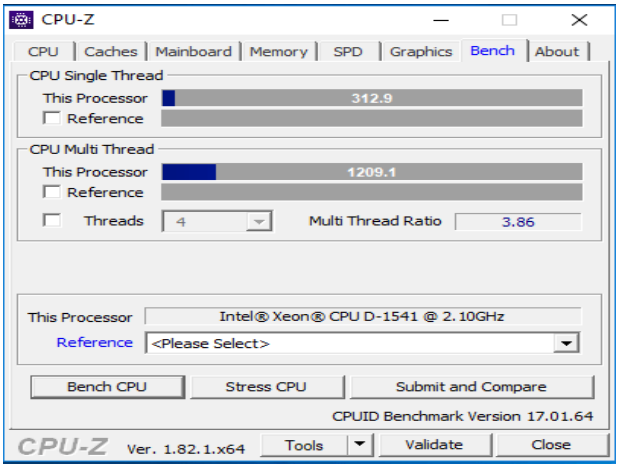
CPU benchmark and Disk benchmark before applying the patch.



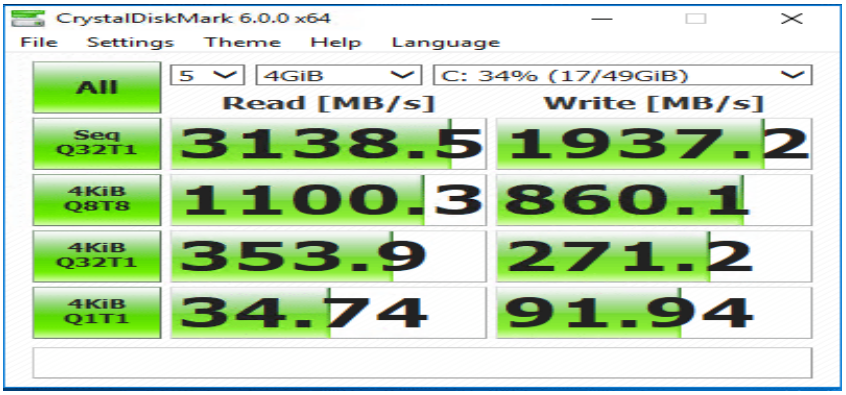

**Images after applying the Microsoft Guest Operating System Patch**

CPU benchmark after applying the Microsoft Meltdown and Spectre patch:



As shown below, the single thread performance is exactly the same. However, the multi thread performance has went down 1.79%.

Disk benchmark after applying the Microsoft Meltdown and Spectre patch:



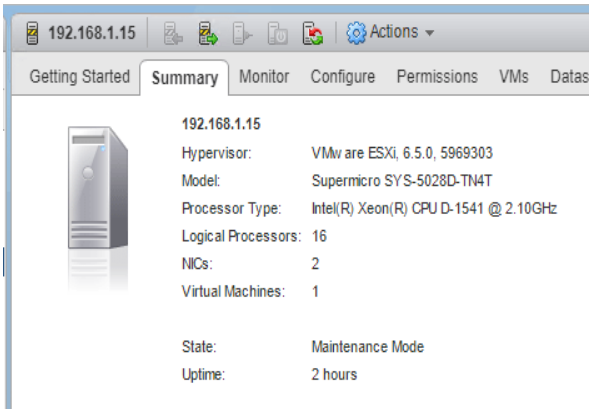

The numbers shown in the crystal Disk, while not wildly different are lower aside from the Seq Q32T1 bench on the read side. But other numbers are lower especially on the write side.

**After Applying the VMware ESXi Meltdown and Spectre Microcode Patch**

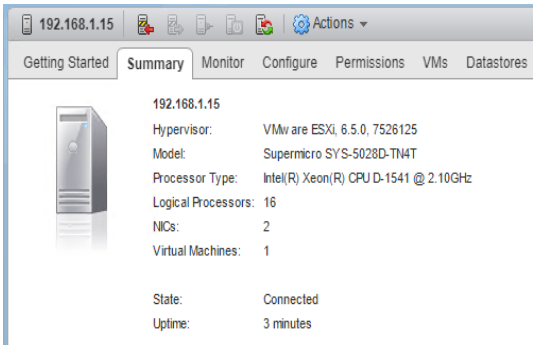
They have applied one of the following ESXi patches to update the microcode for supported CPUs

- ESXi600-201801402-BG microcode *
- ESXi550-201801401-BG hypervisor and microcode **

ESXi 6.5 before patching:

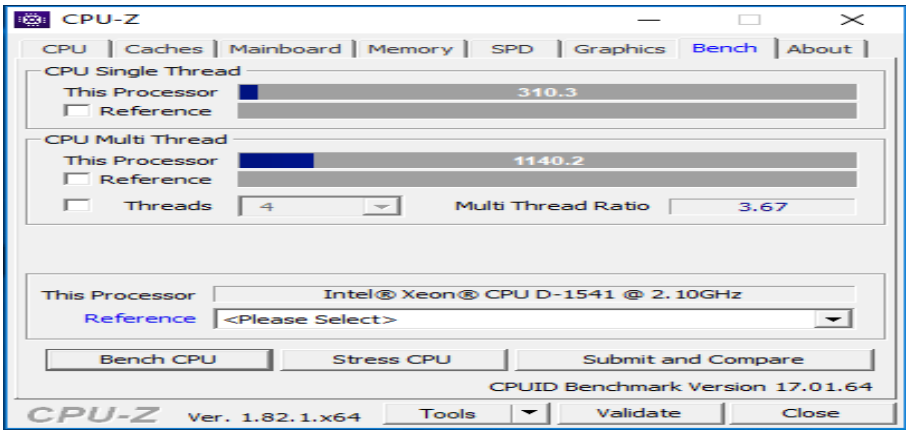



ESXi 6.5 after patching :



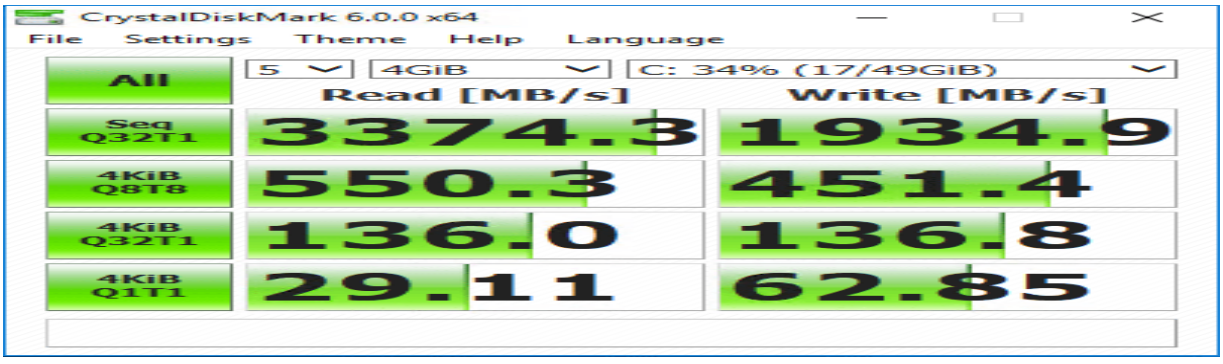

CPU benchmark after applying VMware ESXi Meltdown and Spectre patch:

Here we can see a pretty dramatic decrease in the **Multi Thread** performance as it is a full **7.39% less performance**.



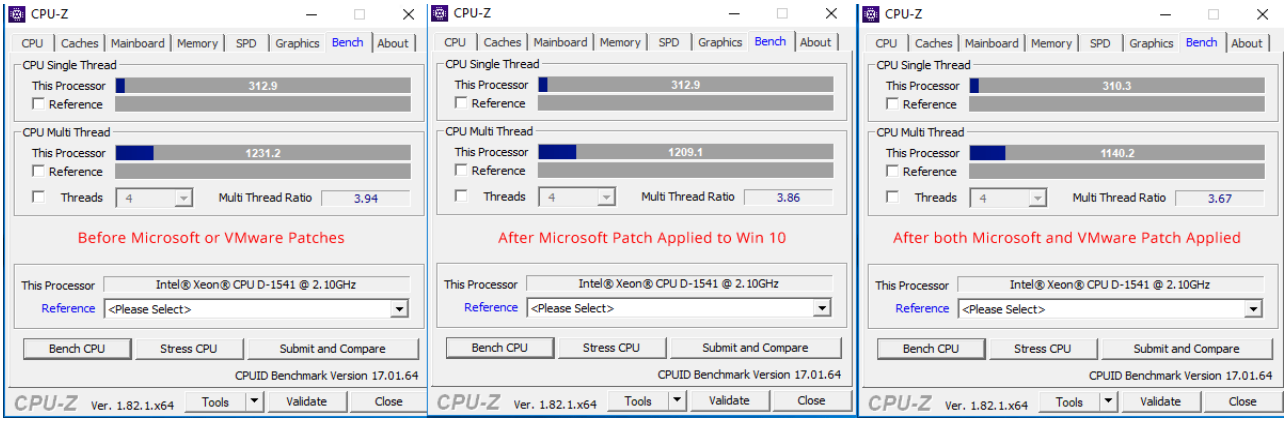

Disk benchmark after applying VMware ESXi Meltdown and Spectre patch:

In the below CrystalDisk benchmarks the **Write MB/s** performance are hit hard as seen below.




Comparison between CPU and Disk performance between Meltdown and Spectre Patches

We can see the results as below, when comparing the various Meltdown and Spectre patches and their effect on performance




There are multiple ways in which overall security can be achieved by hardening the hardware and software layer at different phases of product development life cycle. Best practice would be adhering to Security Development cycle which defines rules to enhance security at each phase of product development (Design, Implementation and Validation). In this survey we have come across one of the major vulnerabilities that existed in the core of the CPU that was dormant for ages till the time it was uncovered and exploited.

The impact of this vulnerability is not limited to processor which is the core of computer hardware, but all the layers of computational system built on top it. Customer machines will always need to be continually updated as more techniques are found that exploit the flaws, such as the one found in processor or any future bugs in the modern computing stack. Software bugs are never going away, cloud provider hypervisors are getting increasingly customized and stripped down to suit each provider's needs and reduce the attack surface area. Patching software is also one of the fundamental solutions to stay secure. But patching has resulted in performance degradation and increased resource usage, as reported for public cloud-based workloads.

From this we can summarize that organizations should invest more time and resources in building

secure hardware and software instead of releasing products which are prone to security attacks and exploitations. Performing intense quality testing with security as top priority would help in identifying the defects and mitigated risk in early stages of the product development.

We should make sure that we don't have a tradeoff between performances and Security. Currently also Spectre hasn't been completely mitigated and there are still chances of some kind of exploitation that could take place.

# 6.Future Work

Meltdown & Spectre also affects Software Defined Storage based implementations in terms of latency and performance. There also exists concern around solutions that use containers to deliver storage resources. They have an impact on container security, enabling one container to read the contents of another for example memory leak between virtual machines in a hypervisor environment.

Major issues that remains open is-

- How data is being protected when storage is being delivered with containers?
- What security implementations would provide protection to ensure a rogue container doesn't get access to all of the data containers on a host?

Hyper-convergence distributes the storage workload across all hosts in a large-scale architecture. Deploying patches for Meltdown and Spectre could increase the storage overhead by up to 50%. This could put some deployments under stress and will certainly impact future capacity planning.

# 7.Conclusion

A fundamental assumption with software security techniques is that the processor will faithfully execute program instructions, including its safety checks. Software security fundamentally depends on having a clear common understanding between hardware and software developers as to what information CPU implementations are (and are not) permitted to expose from computations. The vulnerabilities in this survey, originate from a long-standing focus in the technology industry on performance improvements. As a result, processors, compilers, device drivers, operating systems and many other critical components have developed complex optimization compounding layers that pose security risks.

We presented Meltdown, a software-based attack exploiting out-of-order execution and side channels on modern processors to read arbitrary kernel memory from an unprivileged userspace program. Without requiring any software vulnerability and independent of the operating system, Meltdown enables an attacker to read sensitive data of other processes or virtual machines in the cloud affecting millions of devices. Spectre does not require any software vulnerabilities and

allow attackers to read private memory and register contents from other processes and security contexts. We have seen that the countermeasures provide a short-term goal and improvements must be done at the hardware level to improve performance. Countermeasures provide a short-term protection. As a result, we believe that long-term solutions will require fundamentally changing instruction set architectures.

# 8.References


(1) https://medium.com/@mattklein123/meltdown-spectre-explained-6bc8634cc0c2

(2) https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

(3) https://spectreattack.com/spectre.pdf

(4) https://meltdownattack.com/meltdown.pdf

(5) https://meltdownattack.com/

(6) https://cloud.google.com/security/cpu-vulnerabilities/

(7) https://www.quora.com/What-is-Speculative-Execution

(8) https://danielmiessler.com/blog/simple-explanation-difference-meltdown-spectre/

(9) https://www.csoonline.com/article/3247868/vulnerabilities/spectre-and-meltdown-explained-what-they-are-how-they-work-whats-at-risk.html

(10)http://research.cs.wisc.edu/multifacet/papers/hill_mark_wisconsin_meltdown_spectre.pdf

(11)https://www.ellexus.com/wp-content/uploads/2018/01/180107-Meltdown-and-Spectre-white-paper.pdf

(12)https://en.wikipedia.org/wiki/Side-channel_attack

(13)https://en.wikichip.org/wiki/cve/cve-2017-5715

(14)https://www.csoonline.com/article/3247868/vulnerabilities/spectre-and-meltdown-explained-what-they-are-how-they-work-whats-at-risk.html

(15)https://github.com/speed47/spectre-meltdown-checker

(16)https://software.intel.com/securitysoftwareguidance/apiapp/sites/default/files/337879-analyzing-potential-bounds-Check-bypass-vulnerabilities.pdf?source=techstories.org

(17)https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf

(18)https://www.intel.com/content/www/us/en/architecture-and-technology/facts-about-side-channel-analysis-and-intel-products.html

(19)https://forums.aws.amazon.com/thread.jspa?messageID=822571

(20)https://www.businessinsider.com/google-amazon-performance-hit-meltdown-spectre-fixes-overblown-2018-1

(21)https://kb.vmware.com/s/article/52245