Meltdown and Spectre In the Cloud

Aubhik Mazumdar
Schaefer School of Engineering and Science,
Stevens Institute of Technology
Hoboken, New Jersey 07030
Email: amazumda@stevens.edu

***Abstract*—The Meltdown and Spectre vulnerabilities have proved to be among the most devastating vulnerabilities discovered in the last 20 years. Affecting chips from various manufacturers, these vulnerabilities rely on chip features introduced to improve performance and efficiency namely, speculative execution, branch target prediction and caching. Apart from personal computers, they are also prevalent on hypervisors running in the cloud and guest machines running on these hypervisors. We will attempt to describe the precise effects they will have on the cloud and how they may be mitigated.**

I. Introduction

In early 2017, two vulnerabilities caught the attention of the entire computing world- Spectre and Meltdown. The reason for garnering such massive attention can be attributed to the fact that these are hardware vulnerabilities (exploiting hardware design flaws) that allow programs to steal protected data which is processed in the system kernel. Intel and ARM, two major chip manufactures have addressed this problem and have suggested a radical redesign of their processors, which will substantially impact all their customers. In this report, I will first be describing the flaws that spectre and meltdown exploit, and how they do it. Next, I will be examining the effects of these vulnerabilities on the cloud. The first serious issue pertains to the security of customer data on cloud service providers infrastructures such as Amazon AWS. Currently, cloud providers which use x86 CPUs and Xen PV hypervisors, without patches, are affected. hypervisors that run without real hardware virtualization, relying on containers that share one kernel, such as Docker, LXC, or OpenVZ are also affected. Finally, mitigation procedures are discussed, which may well affect the design of servers and hypervisors being deployed in the cloud. The main flaws in chipsets that allow Meltdown and Spectre to perform are-

1) out-of-order execution
2) speculative execution
3) branch prediction
4) caching in kernel space

Using these, side-channel attacks can be executed by attackers to gain access to data that should be protected and allow code execution on victim systems at the highest privilege level.
Security was once again an afterthought, placing priority on improving performance and efficiency at any cost. This has come back to haunt chip producers with malicious agents exploiting these abilities to read data which should be protected and executing instructions at the highest privilege level. Since these processors are also employed on the cloud as servers the vulnerabilities have been known to affect many cloud providers infrastructures as well. However, compared to personal computers, the dangers are far greater, allowing malicious users on one guest virtual machine (VM) to read protected data of guest VMs running on the same hypervisor. Additionally, an attacker may also read protected data of the hypervisor he/she has access to.
As these vulnerabilities are hardware-based, resolving them completely would require all the chips to change their architectural design, a process that may take years. Software workarounds are the prime mitigation techniques that are being used currently. We will discuss a few of them being employed by Xen Hypervisors and Linux kernels. It is important to note that these mitigations introduce a processing overhead that is non-negligible in the worst case slowing down the system by 30%. Ultimately it seems that the new features did more harm than good.

Contributions:

1) The underlying vulnerabilities in chips that make these vulnerabilities possible are described in a simple, abstracted manner
2) The effects of these vulnerabilities on the cloud are described
3) The latest mitigation techniques on individual hosts and the cloud are discussed, along with their pros and cons.

Outline:

The following sections of this paper first introduce the weaknesses in hardware, in a simple way, that make these vulnerabilities possible. Once we have discussed these flaws a sample proof of concept is described for further explanation of how they are exploited. Next, the effects on the cloud are discussed for each of these vulnerabilities. Finally, we will look at the mitigation strategies in place and will discuss what effects it may have on future solutions.

II. Weaknesses

In this section, I will attempt to simplify the prime weaknesses of processors that have made these vulnerabilities possible.

A. Out-of-order execution

In modern processors, an idle CPU is a wasted CPU. Each CPU cycle must be maximised by assigning it a task when

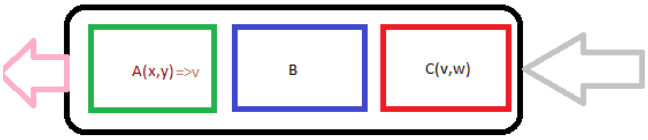

Fig. 1. Out of order execution

it is waiting or lying idle. This is the prime reason out-of-order execution was introduced. In this scheme, independent instructions may be executed if the CPU is lying idle, in any order. All instructions are added to a queue to wait for its operands to become available. As soon as the operands are available, this instruction can be executed even if there are instruction in front of it in the queue. The result of the instruction is remembered and when all the instructions on the queue are executed, they are reordered to make it appear like they were executed in the order written. Figure 1 shows the instruction queue at a point of time. The 3 instructions in the queue are A, B and C which depend on the variables specified in brackets. The in-order execution would be

$$A => B => C \tag{1}$$

The out-of-order execution would be

$$A, B => C \tag{2}$$

As C is a dependent instruction dependent on the result v produced by A. A and B would be executed in parallel as B is an independent instruction. Out-of-order execution begets speculative execution, which we will cover in the next section. Speculative execution is the crux of the problem with these vulnerabilities.

B. Speculative Execution

Put simply, speculative execution allows a processor to execute a branch of a conditional statement before determining whether that branch will be taken. The CPU is directed by the branch predictor which uses many factors like previous execution information to determine which branch must be executed speculatively to save time. In in-order execution, the CPU would lie idle while the conditional statement is being evaluated and only after that will execute the correct branch. Thanks to out-of-order execution, while the conditional statement is being evaluated to judge which branch should be taken, the CPU can already start executing the instructions in one of the branches. After the conditional statement has been evaluated, the CPU will not need to execute the instructions in the correctly predicted branch again, reducing idle time and improving the speed of processing. If the incorrect branch was predicted, the CPU would simply discard the results of the instructions it predicted would be executed and the instructions in the correct branch, the same situation it was in during in-order execution. Thus, the processing speed would be improved in the best case of correct prediction and would be no worse in the case of incorrect prediction. Figure 2 shows an example of this situation in which instructions in the if branch would be executed in parallel with instructions 1 and 2. Once the CPU reaches the if condition, if the branch is taken, the CPU will assign _w and _x to w and x and if the branch is not taken the CPU will simply discard _w and _x and execute the instructions in the else block.

The branch in a conditional statement may contain instructions that access kernel memory or other protected memory regions. During its execution the CPU must use its own cache which leaves a trace of these instructions, a serious problem we will describe in the next section.

C. Caching in Kernel Space

Every CPU possesses a small amount of memory which is used to accelerate the speed of memory access. It takes a while for the CPU to access the random memory of a system when compared to accessing the cache. This is due to the proximity of the cache to the processing unit and the exclusive control the CPU has over it. The CPU makes use of this cache for many operations and it does lead to a significant increase in efficiency. The downside of this is that a trace of this is always left in the system. Since the speeds of access of cache and memory are different we can compare the time it takes to access an address and hence determine its location in the system. This is known as a side-channel attack as direct effects are not exploited rather the side-effects of execution are used to exploit the system. A serious problem occurs in the case of kernel memory; when the CPU accesses protected kernel memory, it may still use the cache to store the value which persists in the cache and the area does not have the same protection mechanisms as the kernel memory. Effectively, it removes the layer of protection that this memory has as programs run by a user cannot access kernel memory and so, in some cases, may be able to determine the contents of the cache.

D. Branch Prediction Unit

The branch prediction unit is used during speculative execution to determine which branch should be executed. It determines this by studying the current situation and other statistics such as how many times the branch was taken recently. There are 2 methods of branch prediction- static branch prediction and dynamic branch prediction. In static

```
1  t = a+b;          |  EXECUTION:
2  u=t+c;            |  T0: t = a+b , _w = e + f;
3  if u>15:          |  T1: u = t+c , _x = _w + g;
4      w = e+f;      |  if u>15:
5      x = w+g;      |      T2: w,x = _w,_x;
6  else:             |  else:
7      w = u+5;      |      T2: w = u + 5;
8      x = w+u;      |      T3: x = w + u;
```

Fig. 2. Speculative execution code

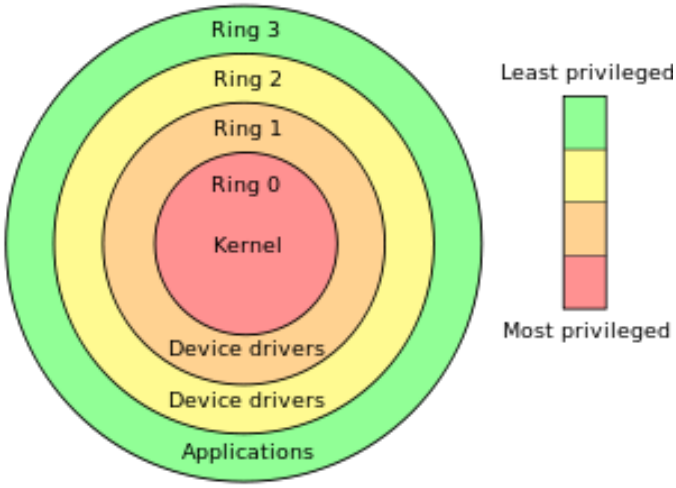

Fig. 3. Protection Rings in x86

branch prediction, the outcome of the prediction is based only on the instruction itself. Dynamic branch prediction on the other hand, gathers statistics at run-time to predict the outcome. In recent developments, neural branch prediction is being employed which relies on neural networks [2] to train the predictors on which branch to take. In the case of Meltdown, mis training the branch predictor to take a branch invariably allows the attacker to execute malicious code in the branch. Spectre takes this further by poisoning the indirect branch predictor and the branch target buffer to execute gadgets in the victims system to leak data.

III. Meltdown

The next few sections discuss the Meltdown vulnerabilty: the attack, dangers, mitigation and the effects it has on the cloud

A. The Attack

Meltdown gets its name from its ability, in a sense, to melt hardware boundaries of the system to gain access to protected data. As noted before, the features of systems that allow it to do this are speculative execution and caching. When data has to fetched from memory, an address has to be specified in the read instruction. The address that is requested has to be validated by the CPU to check whether it is an invalid request. A request may be invalid due to 2 reasons

1) The address is out-of-bounds or does not exist
2) The address exists but the program requesting access to it does not have the correct permissions.

In either case, a fault will be signalled to the processor indicating that the process must be terminated. This is identical in the case of hypervisors where if a guest virtual machine(VM) requests access to an address that belongs to another VM, the hypervisor must issue a fault to prevent it. An address may be inaccessible due to permissions as well. Every process runs in a special mode called user mode. In this mode, the process has its own memory that is mapped to some physical memory in the system. Kernel memory is also mapped but the process running in user mode cannot access it. To access this region the process must enter the kernel mode by issuing a trap or by signalling an interrupt, after which the CPU takes over execution. Some regions of memory are also not accessible due to the privilege level of the process that is trying to access the memory. This is defined in the x86 architecture using Privilege Levels or Protection Rings. There are 4 privilege levels that have different permissions. A program running at privilege level x can access memory for that level and all memory with privilege levels greater than x. Level 0 is hence called the kernel level and level 3 is called the user code level. Processes usually run on level 0 or level 3. Whenever I/O needs to be performed or memory needs to be accessed, the process switches to Kernel mode. Figure 3 shows all the protection rings and the associated processes. Meltdown is responsible for melting exactly these boundaries. This is possible thanks to speculative execution. Consider the code shown in Figure 4 Now, speculative execution makes it possible for the CPU to execute the instructions in the block while expr() is being evaluated. If address is invalid there can be 2 reasons why it is invalid (mentioned above). If the address does not exist, the processor simply queues a fault and waits for expr() to be evaluated. If the branch is taken it will simply signal a fault. The issue is that when the address is invalid due to incorrect permissions, the processor does not know this as the permissions for the access are still being evaluated. Due to this, execution will continue, and the exception raised will be handled after the speculative execution is finished. Thus, the processor will bring the value at address into the cache for processing and will also execute line 5. Once the permission is evaluated, a fault will be signalled but the damage has already been done. Since we loaded a value in memory with the index of the value at address we can just check which index was loaded to determine the value. To check which index was loaded we will execute a side-channel attack. As the value is already in the cache, it will be accessed quicker than the other locations of $C. Using a simple timing comparison, we can find out the value of the memory at address. Figure 3 shows that our cached byte has the value 2. After multiplying it with 4096 we see that index 8192 is marked as hot while the rest are cold as they are not cached. This is a basic description of

```
01 $A = expr()
02 if($A is 1)
03 {
04     $B = load_memory(address)
05     $C = load_memory($B)
06 }
```

Fig. 4. Conditional branch speculative execution demonstration

Meltdown abstracting the technical details. It is similar to the act of knocking on or shaking a box to determine what is inside it. There are also mechanisms of handling these exceptions and making them work for the attacker. There are 2 approaches to this; exception handling and exception suppression.

Exception handling: In exception handling, the exception that is raised due to a reference to inaccessible memory is caught and is not allowed to affect the system. One method of doing this is by forking the attacking application before executing the transient instructions: those instructions that are executed out-of-order leaving side-effects to exploit. By forking the process, the CPU will execute the child process first and will crash on seeing the exception, the parent process can then recover the secret information through a side-channel to gather the data left by the child process. This is called the fork-and-crash approach. Another method involves installing a signal handler that will execute instructions specified by the attacker whenever the exception signal is raised, in our case a segmentation fault. The attacker can thus also prevent the process from crashing reducing the extra work that has to be done when terminating the child process.

Exception handling: Another alternative for handling exceptions prevents the exceptions from ever being signalled to the system. We can do this using transactional memory in which all operations are executed atomically. If any of the instructions cause an exception the state of the machine is rolled back to what it was before the instructions were executed. Finally, speculative execution can be used to suppress exceptions by adding them to a branch that is never actually executed by the CPU. By assuring that the branch is never executed in the code path, we can assure that the memory access will always be executed speculatively and will never cause an exception. Spectre makes extensive use of this.

B. Dangers

Meltdown allows an attacker to dump the entire kernel memory and physical memory of a PC or a virtual machine on the cloud remotely. This can be done by iterating through the entire mapped address space using one of the exception handling or suppression mechanisms to prevent the process from crashing repeatedly. This has been proved on all Windows and Linux operating systems. The processors that were used for these tests include all i-series processors from Intel and Intel Xeon processors in the cloud. Linux has attempted to prevent unauthorized access to Kernel memory by employing kernel address space layout randomization (KASLR) in all versions after kernel 4.12. KASLR randomizes the offset of the device drivers on every boot so the kernel data structures have to be searched for each time. This does make the exploit more difficult to execute, but it is still possible. All that needs to be done is to find the kernel address space by searching through the entire address space. As we see in figure 5, the physical memory is directly mapped in the kernel at a certain offset, the blue region also shows that the user space is also directly mapped in the kernel. Alternatively, an attacker may also just iterate through the virtual memory space linearly.

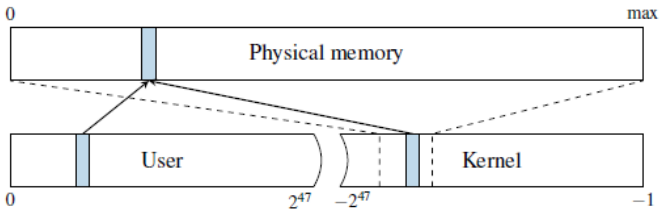

Fig. 5. Memory mapping in User and Kernel space

Windows has the worst repercussions of Meltdown as it maps the kernel into the address space of each application. Also, there is no randomization as the physical memory space is linearly mapped to the virtual memory space. It was also possible to read the binary of the Windows Kernel using Meltdown. Address space layout randomization (ASLR) can also be bypassed to find the exact location of kernel data structures [3,4,5]. Javascript also allows 'derandomization' of ASLR [6]. Sufficient knowledge about the address space combined with the software bug may also allow remote code execution.

C. Mitigation

Even though Meltdown is a relatively new vulnerability, a few mitigation measures have been explored. We will discuss 2 of these methods- Hardware-based prevention, KAISER and Retpoline.

1) Hardware-based prevention: As we know, meltdown is a hardware-based vulnerability taking advantage of the misgivings of the internal system to leak sensitive data. Software solutions can be concocted but they will all have a considerable effect on the performance of the system. Hardware-based solutions will prevent meltdown without the sacrifices in performance but to do this all chip architectures must be changed; a mammoth task considering the number of chips that exist in the world. Let us deal with a trivial solution first; preventing out-of-order execution will indeed prevent this attack but the repercussions on the systems speed will be too great to bear as parallelism will effectively disappear. The next solution could be to serialize the request for accessing memory and the memory access itself. In this method, the processor will wait for the result of the permission check and only then will it allow access to the memory location. Again, memory access will be slowed down as each time the process will halt while the result of the permission check is returned. The best option proposed by the authors of the original paper [7] would be to split the user and kernel space using an extra bit in the CPU control register. This split would ensure that if the bit is set to 1, the kernel addresses lie in the top half of the address space and the user addresses reside in the bottom half. Thus, when memory needs to be fetched, only the requested address needs to be checked to see whether the process has access to it. This will improve performance and improve security.

2) KAISER/KPTI: As hardware-based solutions will be hard to implement in the near future, software workarounds

have been proposed. One such method is KAISER which is also called Kernel Page Table Isolation or KPTI [8]. It was initially introduced in Linux kernels to prevent side-channel attacks that break KASLR but also has the recently discovered advantage of preventing Meltdown. A similar patch has also been released for Windows 10 Build 17035 [9] and for Mac OS X and iOS [10]. Even this method has some limitations. The x86 architecture maps several privileged memory locations to user space [11] thus vulnerable to Meltdown from user space. Pointers can be leaked from these instructions that help defeat KASLR and identify the address of the kernel address space.

3) RETPOLINE: This approach has been developed by Google engineers and its name comes from a mix of return and trampoline [12]. In this method the pointers to the randomized address are protected by using a trampoline location for every kernel pointer. The interrupt handler would not call the kernel code directly but would do it through the trampoline function. This trampoline function must only be mapped in the kernel and randomized.

D. Meltdown in the Cloud

As mentioned before, meltdown has serious effects on the cloud since all virtual machines run the same operating systems that are run on individual PCs. More importantly though, there are more users, more points of attack and shared data. We will consider the three primary parts of the cloud to see how they are affected: servers, hypervisors and containers.

1) Servers: Servers on the cloud see a high throughout of data due to the multiple virtual machines running on them. These servers, that run virtual machines, are more susceptible to attacks as multiple VMs can use it at the same time. Even if one machine on a server is infected, it can access sensitive data of all the other machines running on the server. To protect the servers from a meltdown attack, we will need to implement hardware-based techniques as software based techniques may slow down the server and its effect will be magnified on the cloud.

2) Hypervisors: The middleman between the servers and the VMs is the hypervisor which accepts interrupts from the VMs and performs the necessary actions returning the result to the VM when completed. The hypervisor is the first line of defense against the meltdown attack. It behaves like an operating system for the entire cloud system consisting of VMs and servers and thus should be treated like one. It isolates VMs from each other, performs privileged instructions for the VM and performs translation of addresses. They are susceptible to the same issues that Meltdown exploits and so patches need to be applied to them the same way. Hardware based virtualization (HVM) is seen to be less affected by meltdown than Paravirtualized guests. This is due to the fact that HVM Virtual machines are fully virtualized and thus have no knowledge of other machines. According to Xen, only Intel processors in 64-bit PV mode can attack Xen Hypervisors using Meltdown [13]. Guests running in 32-bit PV mode, HVM mode and PVH mode cannot attack the hypervisor using Meltdown but guest user programs can attack guest kernels. KAISER is the most logical solution that can be used right away to patch the guest kernels. Guest kernels running in 64-bit PV mode are not vulnerable to attack using meltdown because they are already run in a KPTI-like mode. Xen has released its own patches named Vixen and Comet and is still working on alternative solutions. These patches run PV guests in a PVH container which ensures that PV guests continue to behave as before while providing the isolation that protects the hypervisor from Meltdown. It is important to note that performance hits in the hypervisor will be felt in all the VMs and so will be magnified in the cloud.

3) Containers: The authors of the paper on Meltdown have evaluated Meltdown running in containers sharing a kernel, including Docker, LXC and OpenVZ. It was observed that Meltdown can be mounted without any restrictions. In this case, Meltdown can leak information inside the kernel as well as information from all the other containers running on the same physical host. Additionally, the isolation of containers sharing a kernel is totally defeated by Meltdown. This puts cheap hosting providers at risk as they host virtual machines on containers rather than fully virtualized machines. Again, the best solution would be to use KAISER to randomize the kernel address space which is shared among all the containers.

IV. Spectre

Spectre is a name given to the 1st two variants of the speculative execution attacks discovered by Jann Horn and his team at Google (CVE-2017-5753 and CVE-2017-5715). As you may have guessed, it gets its name from the fact that it exploits speculative execution. It also alludes to its English meaning of a ghost or obscure object that is dangerous and cannot be contained. In the next few sections we will see how the attack is performed, what dangers it poses, the mitigation strategies (not many) and the effect Spectre may have on the cloud.

A. The Attack

To mount spectre, the attacker first searches for a sequence of instructions in the process address space which can act as a covert channel transmitter and leak the victims memory or register contents. The attacker then trains the CPU to mis predict and erroneously execute this sequence to create the covert channel and transmit the secret memory. Finally, the attacker can retrieve the memory that was leaked over the covert channel using side-channel attacks. In this way, the victims CPU does not notice anything strange as the speculative execution is simply rolled back but microarchitectural changes persist, which are then exploited. There are 2 techniques for inducing and influencing erroneous speculative execution. The first one exploits conditional branches as described in section

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

Fig. 6. Spectre Attack demonstration

2.2. Let us take a simple example of an 'if' conditional statement. As described in Sec 2.2, Figure 6 shows an example of vulnerable code. The if condition checks whether the index we have requested is in or out of bounds. Initially, the attacker will provide in-bound values of x so that the branch is taken, and the predictor is trained to take the branch. Now, once the branch is trained, the attacker will provide an out-of-bounds value of x to access some memory that the program should not have access to. While the condition is being evaluated the CPU will speculatively execute the instructions in the branch since we have trained it to. This memory value will be stored in the cache as we are assigning it to the variable y. After this is done, the flush + reload technique can be used to determine which index was stored and we can determine the value of array1[x]. The second technique that is used by the researchers is by exploiting indirect branches. In this method, the attacker must train the Branch Target Buffer (BTB) to mis predict a branch from an indirect branch address instruction to the address of the gadget, which is then speculatively executed. A gadget will perform the leak of the sensitive information in the cache and can be used to read arbitrary memory from the victim.

1) Technique 1: Exploiting Conditional Branch Misprediction: Assuming that the code in Figure 6 is a part of a function (kernel syscall or cryptographic library i.e compute SHA) that uses the value x from an untrusted source, the process will have access to array1 of size array1_size and array2 of size 64KB. If the cache structure is known [14] or if the CPU has a cache flush instruction, then the attacker can set the cache state to whatever he desires. After this, the attacker executes the code with valid values of x for a few iterations. The attacker then sets x to a value that is out-of-bounds to read the secret information k. The value of k is then used to compute the address of array2[k*256]. To recover the secret information the attacker can use the same technique used in Meltdown by comparing the times of access for all the elements of array2 through which the fastest returned index will represent k. The attacker may also use the prime-and-probe attack [15] in which the attacker detects which index was removed from the cache on the latest read.

The technique was used on multiple processor architectures including the Intel Skylake (an unspecified Xeon on the Google Cloud), AMD Ryzen and the Intel Ivy Bridge. Spectre was observed on all these processors in 32-bit and 64-bit mode and Linux and Windows. Some ARM processors were also found to be vulnerable [16].

The Spectre vulnerability is extremely dangerous due to the sheer number of intructions that can be executed speculatively. Researchers found that on an i7 processor, 188 simple instructions could be executed between the 'if' statement and the line accessing array1/array2.

The researchers have written a proof of concept in two languages C and Javascript. The C code when compiled first trains the branch predictor in victim_function() to always execute the branch using the read_memory_byte() function. After this, the victim_function() passes an out-of-bound x value which the conditional branch mispredicts and the next

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

Fig. 7. Speculative execution in Javascript

instruction reads a secret byte to get the secret value. The code then reads array2[array1[x] * 512] which adds the value of array[x] into the cache. Finally, a flush+probe is executed to find the cache line that was loaded and collect its contents. The attack is repeated so that even if the first iteration does not add the target value to the cache, the next iteration will. The code can read data at approximately 10KB/second. The Javascript implementation allows a code running in the browser to read private memory of the process which is running it. It is interesting to note that the Google Chrome and Mozilla Firefox browsers create a new process for each tab that is opened and hence are more vulnerable to the attack than Internet Explorer or Safari.

The values of TABLE1_STRIDE and TABLE1_BYTE is first set to 4096 and 225 respectively, an implementation detail. During branch training the index is set an in-range value by performing bitwise operations with TABLE1_STRIDE and TABLE1_BYTE. On the final iteration, index is set to an out-of-bounds address into simpleByteArray. The localJunk variable ensures that these operations are not removed by the interpreter in lieu of optimizing the code. The OR bitwise operations are also only to ensure that the interpreter treats these values as integers. The researchers disassembled the code using the D8 tool to set the length of the simleByteArray in memory manually rather than it being cached in a register. Next, the cache had to be flushed which was done by reading a series of addresses at 4096 -byte intervals out of a large array. The leaked information was gathered using the probeTable array by reading the cache status of probeTable[n*4096] for all n between 0 and 255.

Timing comparison in JavaScript is much tougher as there is no rdtscp (timestamp counter) instruction and Chrome tries to obscure the values of timing data to prevent timing attacks using performance.now() [17]. HTML5 though provides the web workers feature which makes it simple to create a separate thread that can act as a high-resolution timer sufficient for our purpose. [18,19]

2) Indirect Branch Instructions Poisoning: Indirect instructions can jump to more than 2 possible target addresses like an address in a register or an address in memory location or an address stored in the stack (RET). If the destination of the jump is not in the cache, and the branch has been trained to take the jump invariably, speculative execution may continue at a location chosen by the attacker. In this manner, the attacker can cause the program to be misdirected to locations that would never occur during normal execution. Attackers can thus expose memory locations even in the absence of a conditional branch statement.

For example, an attacker wants to read the memory contents of a victim by manipulating two registers it has under its

control (R1 and R2) when an indirect branch occurs. A function call may use these two registers and pushes them on the stack at the beginning of the function call and removed at the end. The attacker can next use 2 gadgets one after the other by first adding, subtracting, XORing the memory location addressed by R1 into R2 followed by an instruction that reads the memory contents addressed in R2. It is not necessary for the attacker to control two registers. The same effects can be achieved using different gadgets operating on a single register, value on the stack or memory value.

B. Variations on Spectre

Spectre does not need to rely on effects on the cache of the CPU, there are many other microarchitectural effects of speculative execution that can be studied once we have the ability to execute gadgets.

Evict+Time: In this variation the timing of operations that depend on the state of the cache are measured to find out the secret value. A simple read statement after the conditional block can tell us whether the variable used in the branch was in the cache or not. If the read in the block was a cache miss, then the second read would take longer. If the read was not a cache miss, the second read would complete quickly. In this case, the contents of the cache do not need to be modified, all that is required is that the state of the cache affects the timing of speculatively executed code.

Instruction Timing: We may also just use certain instructions whose timing depends on the location of the operands in the instruction. [20] Once again we can execute 2 multiplication operations consecutively, on registers R1, R2, R3 and R4. Depending on when the second multiply instruction was executed, we can determine the location of the operands of the first multiply instruction.

Without Conditional Branches: Code without conditional branches may also be vulnerable to Spectre. If the attacker wants to find whether the register R1 contains a value X he can do so by mistraining the branch predictor such that, after an interrupt occurs, the interrupt returns to an instruction that reads the value of memory[R1]. The attacker then chooses a suitable X to correspond to a memory address suitable for a flush+reload.

C. Mitigation

The conditional branch vulnerability can be mitigated by stopping speculative execution on sensitive paths. Serializing instructions on the Intel x86 appears to do well but it only guarantees that the instructions are discarded if speculatively executed. This is not enough as it does not ensure that speculative execution will not occur or leak any information. Next, the compiler could be instructed to insert delaying instructions before and after every conditional branch, but this would considerably affect performance. Also, legacy systems would have to be recompiled which would not be easy. Indirect branch poisoning is even more tricky to mitigate in software. Hyperthreading may be disabled and the branch prediction history may be flushed regularly but there is no existent

Exploited Vulnerability	CVE	Exploit Name	Public Vulnerability Name	Windows Changes	Silicon Microcode Update ALSO Required on Host
Spectre	2017-5753	Variant 1	Bounds Check Bypass	Compiler change; recompiled binaries now part of Windows Updates Edge & IE11 hardened to prevent exploit from JavaScript	No
Spectre	2017-5715	Variant 2	Branch Target Injection	Calling new CPU instructions to eliminate branch speculation in risky situations	Yes
Meltdown	2017-5754	Variant 3	Rogue Data Cache Load	Isolate kernel and user mode page tables	No

Fig. 8. Microsoft mitigation mechanisms

method for doing this currently [21]. Also, all cases may not be covered in this research like the switch() statement and other jump statements response to speculative execution and hence further research needs to be carried out. Google engineers have surmised that the only way to avoid issues with Spectre is to avoid generating code which contains an indirect branch that could have its prediction poisoned by an attacker. The retpoline construct has been proposed to implement indirect calls in a non-speculative way. It can be thought of loosely as a trampoline for indirect calls which uses the RET instruction in x86 processors. After this, we need to arrange for a specific CALL and RET sequence which ensures the processor predicts the return to go to a controlled, known location. The retpoline then damages the return address pushed onto the stack by the call with the desired target of the original indirect call. The result is a predicted return to the next instruction after a call (which can be used to trap speculative execution within an infinite loop) and an actual indirect branch to an arbitrary address. [23] Microsoft has also released a patch for its Visual C++ compiler (MSVC) but has not discussed the technical details of how the attack is being mitigated. [25] The recommended suggestions by Microsoft are shown in figure 8. To summarise, it is extremely difficult to mitigate the Spectre vulnerability in software due to the points mentioned above. The only safe way to guarantee that Spectre can be eradicated is to stop processors from leaving any trace of speculative execution. One such way may be to use a register that cannot be accessed by any program and is implicitly addressed during speculative execution. Using speculative execution injudiciously was a terrible idea and has lead us to this situation.

D. Effects on the Cloud

The Google Project Zero research on this topic clearly states that Spectre will have extreme repercussions on the cloud. Specifically, variant 2 of Spectre allows bypassing privilege and other security boundaries such as those between a virtual machine and the hypervisor running the virtual machine. Figure x shows how this is possible even though the attack is extremely complicated and requires the attacker to have extensive knowledge about the architecture of the hypervisor and the physical machine it is running on. ”The basic idea for

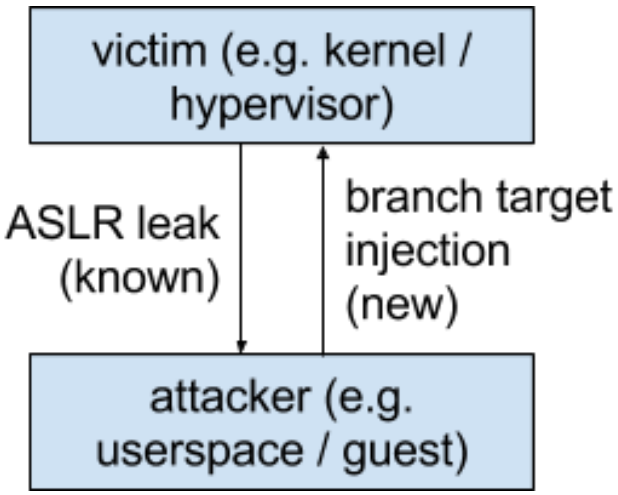

Fig. 9. Spectre on the cloud

the attack is to target victim code that contains an indirect branch whose target address is loaded from memory and flush the cache line containing the target address out to main memory. Then, when the CPU reaches the indirect branch, it won't know the true destination of the jump, and it won't be able to calculate the true destination until it has finished loading the cache line back into the CPU, which takes a few hundred cycles. Therefore, there is a time window of typically over 100 cycles in which the CPU will speculatively execute instructions based on branch prediction." [22] The Intel Xeon Haswell processor was studied and found to be vulnerable. The steps involved in the exploit to read host memory from a Linux KVM (Kernel-based Virtual Machine) hypervisor guests are described below. KVM is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko. Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc. The attacker first needs to determine 3 things: the lower 20 bits of the address of kvm-intel.ko, the full address of kvm.ko and the full address of vmlinux. Vmlinux is a static executable file in Linux systems which contains the Linux kernel specified in one of the Linux supported object file formats like ELF, COFF and a.out. This needs to be extracted and disassembled before it can be used by the system. Each virtual machine will have its own vmlinux file. In the first step, the address of the kvm-intel.ko is guessed by dumping out the entries of the guests branch history table (which stores the recently taken indirect jumps) and then guessing every possible value for bits 12 to 19 of the load address of kvm-intel.ko. The contents of the branch history table are leaked by comparing the misprediction rates of an indirect call with two targets. If the target was in the history table, the jump would be faster. A vmcall instruction is used for calling indirect functions followed by N branches with relevant source and target address bits all zeroes. With N=29 mispredictions occur at a high rate as all the values in the branch history table have been erased and replaced with zeroes. For N=28, mispredictions occur if the controlled branch history value is 0,1,2 or 3. All possibilities have to be tested by decreasing N till it reaches 0 to determine the branch history value for N=0. Now that we know the low 20 bits of the kvm-intel.ko, we need to find the rough location of kvm.ko. Whenever a hypercall is issued, an indirect call from kvm.ko to kvm-intel.ko is used which in turn inserts data into the Branch Target Buffer (BTB). This means that the source address of the indirect call needs to be leaked out of the BTB to get the address of kvm.ko. The last information we need is the load address of vmlinux, which we can determine in a similar fashion by using an indirect call from vmlinux to kvm.ko. In this case, the hypercall will not actually cause an indirect call to kvm.ko and so we need to use port I/O from the status register of an emulated serial port in the virtual machine. The Intel Xeon Haswell processor was studied and found to be vulnerable. The steps involved in the exploit to read host memory from a Linux KVM (Kernel-based Virtual Machine) hypervisor guests are described below. KVM is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, kvm.ko, that provides the core virtualization infrastructure and a processor specific module, kvm-intel.ko or kvm-amd.ko. Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: a network card, disk, graphics adapter, etc. The attacker first needs to determine 3 things: the lower 20 bits of the address of kvm-intel.ko, the full address of kvm.ko and the full address of vmlinux. Vmlinux is a static executable file in Linux systems which contains the Linux kernel specified in one of the Linux supported object file formats like ELF, COFF and a.out. This needs to be extracted and disassembled before it can be used by the system. Each virtual machine will have its own vmlinux file. In the first step, the address of the kvm-intel.ko is guessed by dumping out the entries of the guests branch history table (which stores the recently taken indirect jumps) and then guessing every possible value for bits 12 to 19 of the load address of kvm-intel.ko. The contents of the branch history table are leaked by comparing the misprediction rates

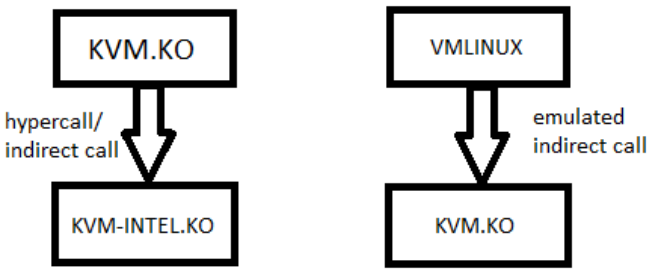

Fig. 10. Exploiting a Kernel Virtual Machine

of an indirect call with two targets. If the target was in the history table, the jump would be faster. A vmcall instruction is used for calling indirect functions followed by N branches with relevant source and target address bits all zeroes. With N=29 mispredictions occur at a high rate as all the values in the branch history table have been erased and replaced with zeroes. For N=28, mispredictions occur if the controlled branch history value is 0,1,2 or 3. All possibilities have to be tested by decreasing N till it reaches 0 to determine the branch history value for N=0. Now that we know the low 20 bits of the kvm-intel.ko, we need to find the rough location of kvm.ko. Whenever a hypercall is issued, an indirect call from kvm.ko to kvm-intel.ko is used which in turn inserts data into the Branch Target Buffer (BTB). This means that the source address of the indirect call needs to be leaked out of the BTB to get the address of kvm.ko. The last information we need is the load address of vmlinux, which we can determine in a similar fashion by using an indirect call from vmlinux to kvm.ko. In this case, the hypercall will not actually cause an indirect call to kvm.ko and so we need to use port I/O from the status register of an emulated serial port in the virtual machine.

V. Conclusion

The vulnerabilities discussed in this paper are relatively new, having been discovered in January 2018. The Meltdown vulnerability was discovered independently and reported by 3 teams Jann Horn (Google Project Zero), Werner Haas,Thomas Prescher (Cyberus Technology) and Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology). Spectre was also independently discovered and reported by 2 people Jann Horn (Google Project Zero) and Paul Kocher in collaboration with Daniel Genkin (University of Pennsylvania and University of Maryland), Mike Hamburg (Rambus), Moritz Lipp (Graz University of Technology), and Yuval Yarom (University of Adelaide and Data61). Jann Horn was particularly impressive as he had managed to discover and exploit the vulnerabilities all on his own considering that he was only 22 years old, a fact that astounded the other researchers. The discovery of these vulnerabilities should act as a warning to all chip producers to thoroughly examine the features that are added to improve performance. Security should not be an afterthought but a priority when releasing new features. These vulnerabilities have shocked the entire world of computer science and are strikingly similar to the heartbleed (CVE-2014-0346) vulnerability discovered in 2014. The scale of its effect is staggering ranging from personal computers, mobile devices, and crucially, servers in the cloud. Mitigation measures are slowly being released in the form of software patches, but as mentioned before, they may affect performance. Hardware-based vulnerabilities are rare but when they are found have greater repercussions than software vulnerabilities. A massive redesign of chips is in order and manufacturers must be pressurized to do so as soon as possible.

References

[1] "Out-of-order execution" (PDF). cs.washington.edu. 2006. Retrieved 17 January 2014. "don't wait for previous instructions to execute if this instruction does not depend on them"

[2] Vintan, L. N., and Iridon, M. Towards a high performance neural branch predictor. In Neural Networks, 1999. IJCNN99. International Joint Conference on (1999), vol. 2, IEEE, pp. 868 873.

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

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

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

[6] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giufridda, C. ASLR on the Line: Practical Cache Attacks on the MMU. In NDSS (2017).

[7] LWN. The current state of kernel page-table isolation, Dec. 2017.

[8] Ionescu, A. Windows 17035 Kernel ASLR/VA Isolation In Practice (like Linux KAISER)., 2017.

[9] Levin, J. Mac OS X and IOS Internals: To the Apples Core. John Wiley & Sons, 2012.

[10] 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. 161176.

[11] https://support.google.com/faqs/answer/7625886

[12] https://blog.xenproject.org/2018/01/04/xen-project-spectremeltdown-faq/

[13] Yarom, Y., Ge, Q., Liu, F., Lee, R. B., and Heiser, G. Mapping the Intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. http://eprint.iacr.org/2015/905.

[14] Osvik, D. A., Shamir, A., and Tromer, E. Cache attacks and countermeasures: The case of AES. In Topics in Cryptology CT-RSA 2006 (Feb. 2006), D. Pointcheval, Ed., vol. 3860 of Lecture Notes in Computer Science, Springer, Heidelberg, pp. 1 20.

[15] Cortex-A9 technical reference manual, Revision r4p1, Section 11.4.1, 2012.

[16] Security: Chrome provides high-res timers which allow cache side channel attacks. https://bugs.chromium.org/p/chromium/issues/detail?id=508166.

[17] Gras, B., Razavi, K., Bosman, E., Bos, H., and Giuffrida, C. ASLR on the line: Practical cache attacks on the MMU, 2017. http://www.cs.vu.nl/ giuffrida/papers/ anc-ndss-2017.pdf.

[18] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. Fantastic timers and where to find them: high-resolution microarchitectural attacks in JavaScript. In International Conference on Financial Cryptography and Data Security (2017), Springer, pp. 247267.

[19] Andrysco, M., Kohlbrenner, D., Mowery, K., Jhala, R., Lerner, S., and Shacham, H. On subnormal floating point and abnormal timing. In 2015 IEEE Symposium on Security and Privacy (May 2015), IEEE Computer Society Press, pp. 623639.

[20] Ge, Q., Yarom, Y., and Heiser, G. Your processor leaks information - and theres nothing you can do about it. CoRR abs/1612.04474 (2017).

[21] https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html

[22] http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html

[23] https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/

[24] https://cloudblogs.microsoft.com/microsoftsecure/2018/01/09/understanding-the-performance-impact-of-spectre-and-meltdown-mitigations-on-windows-systems/