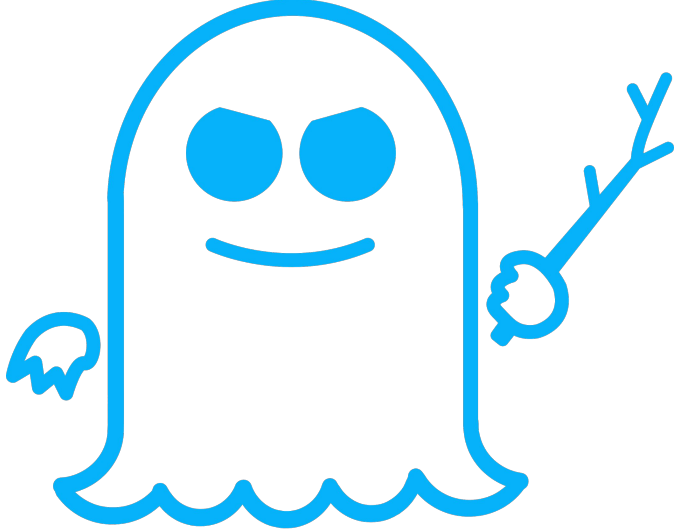
Meltdown and Spectre

Yuhao Jiang, Daiqi Guo

```
meltdown:
mov al, byte [rcx]
shl rax, 0xc
jz meltdown
mov rbx, qword [rbx + rax]
```

MELTDOWN



SPECTRE

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

What are meltdown and spectre?

They are the nicknames for the three vulnerabilities:

- Variant 1: bounds check bypass (CVE-2017-5753)
- Variant 2: branch target injection (CVE-2017-5715)
- Variant 3: rogue data cache load (CVE-2017-5754)

Where Variant 1 & 2 are Spectre and Variant 3 is Meltdown

What do they affect?

- Affects some/all modern processors, servers, mobile phones (Apple SoCs)

- Meltdown
  - Intel, ARM, IBM ...
  - Desktop, Laptop, and Cloud computers
- Spectre
  - Intel, AMD, ARM, IBM ...
  - Desktops, Laptops, Cloud Servers, as well as Smartphones
- Affects all operating systems
  - Linux, Windows, MacOS ...

What do they affect? (cont.)

- Meltdown:
  - Breaks the most fundamental isolation between user applications and the operating system.
  - Allows a program to access the memory, and thus also the secrets, of other programs and the operating system.

- Spectre:
  - Breaks the isolation between different applications.
  - Allows an attacker to trick error-free programs, which follow best practices, into leaking their secrets.

What do they exploit?

- Exploit the three major designs in modern processors:
  - Out-of-order Execution
  - Speculative Execution
  - Caching

- Both attacks use side channels to obtain the information from the accessed memory location.

What is Out-of-order Execution?

- It is an approach to processing that allows instructions for high-performance microprocessors to begin execution as soon as their operands are ready.
- Although instructions are issued in-order, they can proceed out-of- order with respect to each other.
- The goal of OoO processing is to allow the processor to avoid a class of stalls that occur when the data needed to perform an operation are unavailable.

Out-of-order Execution steps

1. Instruction fetch.
2. Instruction dispatch to an instruction queue (also called instruction buffer or reservation stations).
3. The instruction waits in the queue until its input operands are available. The instruction is then allowed to leave the queue before earlier, older instructions.
4. The instruction is issued to the appropriate functional unit and executed by that unit.
5. The results are queued.
6. Only after all older instructions have their results written back to the register file, then this result is written back to the register file. This is called the graduation or retire stage.

Life example of Out-of-order Execution: make tea

Life example of Out-of-order Execution: make tea

- wash tea cups -> boiling water -> make tea
- wash tea cups ----------->-------|
  |-->wait for use------>-----|---->make tea
  boiling water -> boiled->|
- wash tea cups ----------->break cups-->--|
  |-->wait for use-->----------|-->water not use
  boiling water -> boiled->|

- Because the cups are broken when washing them (raise error), the boiled water won't be used in next steps.
- However, don't use the boiled water doesn't mean the boiled water will disappear, it is still placed in the waitting area (caching).

What is Speculative Execution?

- It is a technique used by modern CPUs to speed up performance.
  The CPU may execute certain tasks ahead of time, "speculating" that they will be needed and complete them.
- If the tasks are required, a speed-up is achieved, because the work is already complete.
- If the tasks are not required, changes made by the tasks are reverted and the results are ignored.

Life example of Speculative Execution: order coffee

Life example of Speculative Execution: order coffee

- Barista: make Latte || speculate:need Latte ->make Latte->available Latte---|
  Customer: need Latte || need Latte -|-> Got it !
  Days: day 1 || day 2

- Barista: make Latte || speculate Latte || speculate Latte - make it -|-> make French Vanilla
  Customer: need Latte || need Latte || need French Vanilla---------| & throw away Latte
  Days: day 1 || day 2 || day 3

Caching

- The CPU requests data from memory which is stored in a cache
- Speeds up memory access
- Temporal locality: something which was accessed recently from memory might be accessed again soon
  - Ex. a counter in a loop
- Spatial locality: something which is close to another thing which was accessed recently might be accessed soon
  - Ex. elements in an array

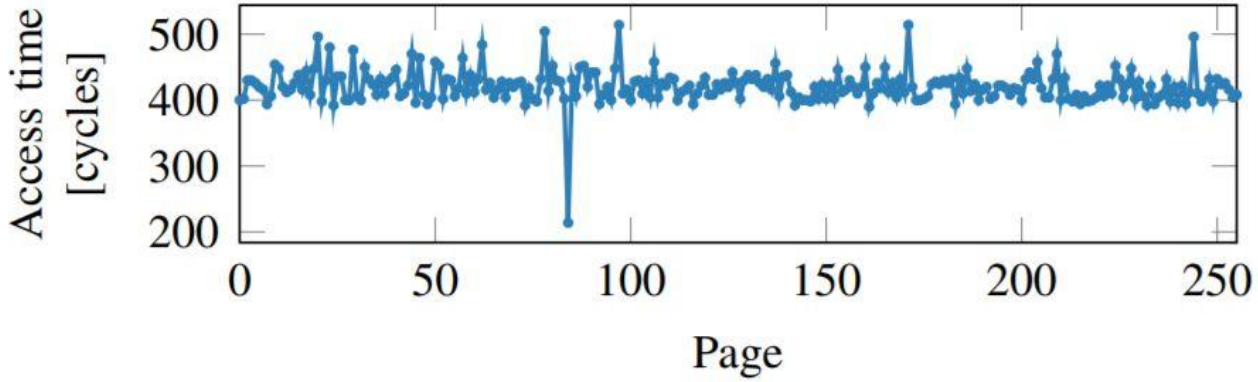
What is side-channel attack?

- Attack which is enabled by the micro architectural design of the CPU and based on information gained from the implementation of a computer system.
  - Caches: attack which monitors how quickly data accesses take and infer whether or not said data was in the cache
  - Timing: attack which monitors time it takes for machine to do various computations
  - Power-monitoring: attack which monitors power consumption of hardware on varius computations
  - ……

Cache side-channel attack

The side channel comes from monitoring how quickly data can be accessed from the cache.

- Data which is accessed quickly => stored in the cache
- Data which is accessed slow => stored in main memory



Exploit Caching

- Flush + Reload
  - Flush any access of memory for data you control from the cache (by clflush)
  - Lets malicious (or user program) run and access memory you control with secret
  - Try reloading elements from the controlled memory and see how quickly they are accessed
- Evict + Reload
  - Evicting memory access of data you control by loading other (possibly random) data into the cache
  - Due to limited size of cache evict the specific cache line
  - Let victim program run and access memory using secret, reload data and measure access time

Privilege check

- Modern CPUs enforce a privilege check of a program accessing kernel memory
  - This privilege check sometimes occurs too late during speculative execution. (i.e. once the data has already been read).
  - Priviledge check aren't performed until the instruction is completed.
- The CPUs knows that this occurs so anything unprivileged which was executed will be forgotten and an exception will be raised (usually SIGSEGV)
  - As a result, the memory that accessed recently is still stored in cache

Meltdown attack

This is the core of Meltdown, let's walk through it step by step.

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

Meltdown attack

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Step 1, first of all, allocates a block of memory consisting of 256 pages of memory (256 * 4096 bytes), we denote it as RBX here.
- Each page in this block of memory won't be cached at this point because it has never been accessed.

Meltdown attack

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Step 2, line 2: xor rax, rax
- This step is used for empty the register rax with all zeros.

Meltdown attack

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Step 3, line 4: mov al, byte [rcx]
- Load the byte value located at the target kernel address which stored in the register RCX, into the least significant byte of the register RAX represented by AL.

Relationship between register AL and RAX

0x1122334455667788

================ rax (64 bits)

======== eax (32 bits)

==== ax (16 bits)

== ah (8 bits)

== al (8 bits)

Meltdown attack

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Step 4, line 5: shl rax, 0xc
- shift left the content in RAX with 12 bits, in another word, (value in RAX)*(2^12) => (value in RAX)*4096
- 4096 => 4096B => 4KB, the size of a page

Meltdown attack

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Step 5, line 6: jz retry
- If we copied nothing into the register AL, the register RAX keeps all zeros, then retry this loop until we copied something into the register AL and make register RAX contain something.

Meltdown attack

```
1 ; rcx = kernel address, rbx = probe array
2 xor rax, rax
3 retry:
4 mov al, byte [rcx]
5 shl rax, 0xc
6 jz retry
7 mov rbx, qword [rbx + rax]
```

- Step 6, line 7: mov rbx, qword [rbx + rax]
- Copied the value into the probe array RBX at index RAX
- Use that multiplied value as an index into the block of allocated memory and read one byte (ie: read one byte from page N where N is the value in RAX)

Meltdown attack

- Assuming the CPU starts out-of-order execution
  - There is a race condition between the privilege check of line 4 codes and the codes after line 4
  - The privilege check may finished after line 5 code.
  - It will cause one page of the allocated block of memory to be cached on the CPU.
- The page that is cached will be directly related to the byte read from kernel mode memory.
  - For example: if the value of the one byte from kernel address is 21, then the 21st page of the allocated memory block will now be cached on the CPU.

Meltdown attack

- Finally, the attacker observes the side effects of this out-of-order execution to determine the secret byte that was read.
  - Catch the exception thrown by privilege check from line 4 code above.
  - Loop through every page in the allocated block of memory
  - Time how long it takes to read one byte from each page.
  - If the byte loads quickly then the page must have been cached and gives away the secret.
- Continuing the example from previous slide, pages 0 through 20 of the allocated memory block would be slow to read, but page 21 would be considerably faster—so the secret value must be 21.

Spectre attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- Exploiting Conditional Branch Misprediction
- The code above is an example of conditional branch
- x = (address of a secret byte to read) - (base address of array1)

Spectre attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- This code looks normal and correct
- If x is less than the length of array1, the loop executes successfully
- But let's assume that we have a variable here that stores the password at the address secret, and let A=secret-array1, so we can use array1[A] to represent the value of secret.

Spectre attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- When the x satisfy the loop condition and we execute this loop for multiple times, the branch predictor will think the next loop also satisfies the loop condition and execute this loop.

Spectre attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- If at this time we assigned the value A to the x, the branch predictor will predict the loop for execution (actually should not execute), the CPU will execute the loop body, and then load the password secret value in cache, and use it as the address to access array2.
- But eventually, the CPU will found this loop should not be executed, so the value got in this loop will become invalid.

Spectre attack Variant 1

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

- Finally, we can read array2, and if we read an address for a short amount of time, that address is the one that is cached (our password value).

Spectre attack Variant 2

- Poisoning Indirect Branches
- Indirect Branch: jumping to code at some memory location
  - e.g. jmp [eax] => jump to instruction stored at memory address in register EAX
- Variant 2 is much like variant 1, but instead of abusing the data lookup portion of the CPU, it abuses the ability for a CPU to predict which way it will go when a function pointer is called.
- The attacker needs to locate a “Spectre gadget”, i.e., a code fragment whose speculative execution will transfer the victim’s sensitive information into a covert channel.

Spectre attack Variant 2

- Attacker chooses a “Spectre gadget” from the victim’s address space and trains the Branch Target Buffer (BTB) to mispredict a branch from an indirect branch instruction to the address of the gadget, resulting in speculative execution of the gadget.
  - Not reliant on the vulnerability of victims code.
  - Attacker has to find the virtual address of gadget
- Exploiting Branch Target Buffer (BTB)

Branch Target Buffer (BTB)

- The Branch Target Buffer (BTB) keeps a mapping from addresses of recently executed branch instructions to destination addresses .

- Processors can use the BTB to predict future code addresses even before decoding the branch instructions.
    - Using Speculative Execution to improve the performance

- Only the 31 least significant bits of the branch address are used to index the BTB.

Branch Target Buffer (BTB)

- Allows the CPU to speculatively execute code at predicted indirect branch target without actually having decoded the branch instructions

- Attacker trains the Branch Target Buffer (BTB) to mispredict a branch from an indirect branch instruction to the address of the gadget.

Code example of Branch Target Buffer misprediction

```c
/*
    BRANCH-TARGET PREDICTION
*/

#include <stdio.h>

void (*print_ptr)();

void print_a() {
    printf("a\n");
}

void print_b() {
    printf("b\n");
}

void print_a_or_b() {
    print_ptr();
}

int main() {
    // teach cpu to predict print_a() on line 17
    print_ptr = print_a;
    print_a_or_b();

    // cpu will incorrectly speculatively predict print_a() before actually doing print_b()
    print_ptr = print_b;
    print_a_or_b();
}
```

Spectre attack Variant 2

- As a result, the gadget code was run by speculative execution because of branch misprediction
  - The result will be loaded into the cache
  - Use cache side-channel attack to gain the secert value

Mitigation

- Anyway, the best way to solve these hardware vulnerabilities is through the hardware way, i.e. re-designing the CPU. However, it may take a lot time and cost huge amount of money.

- Not all computer users will have the money, time or skills to change the computer CPU.

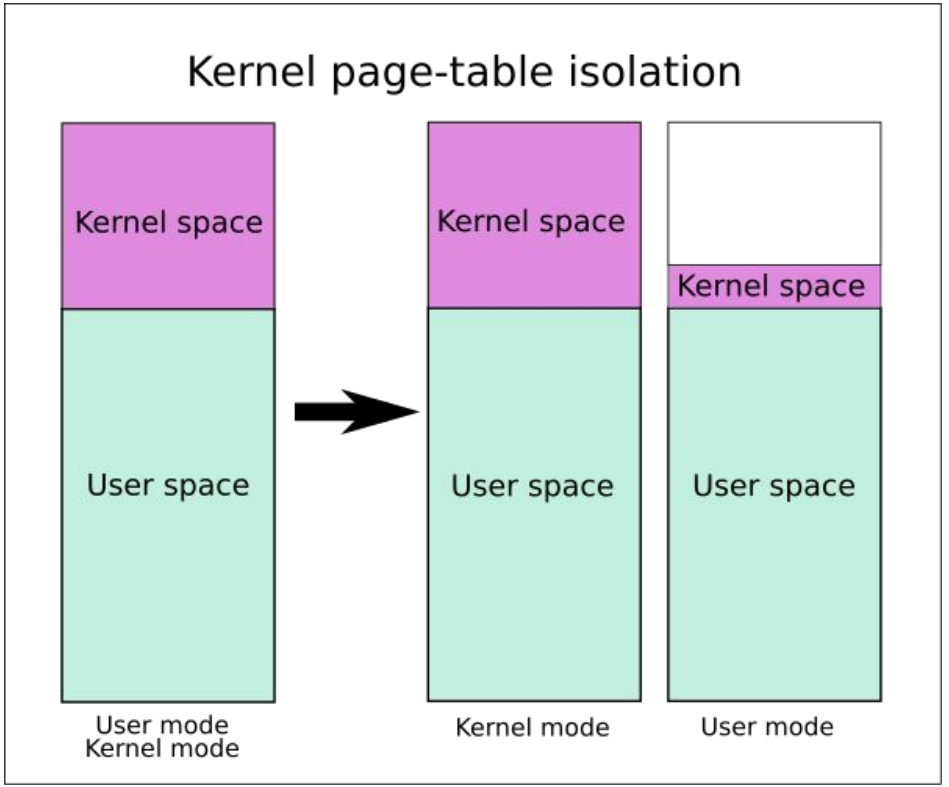
- So, there come out some software patches to mitigate these vulnerabilities through a software way.

Meltdown Mitigation

- Luckily, there are software patches against Meltdown.
- So, update your Operating System and Softwares to the newest version!
- For Linux, this software patch is called KPTI (formerly KAISER)
  - Kernel page-table isolation
  - Kernel address isolation to have side-channels efficiently removed
  - Still have time punishment

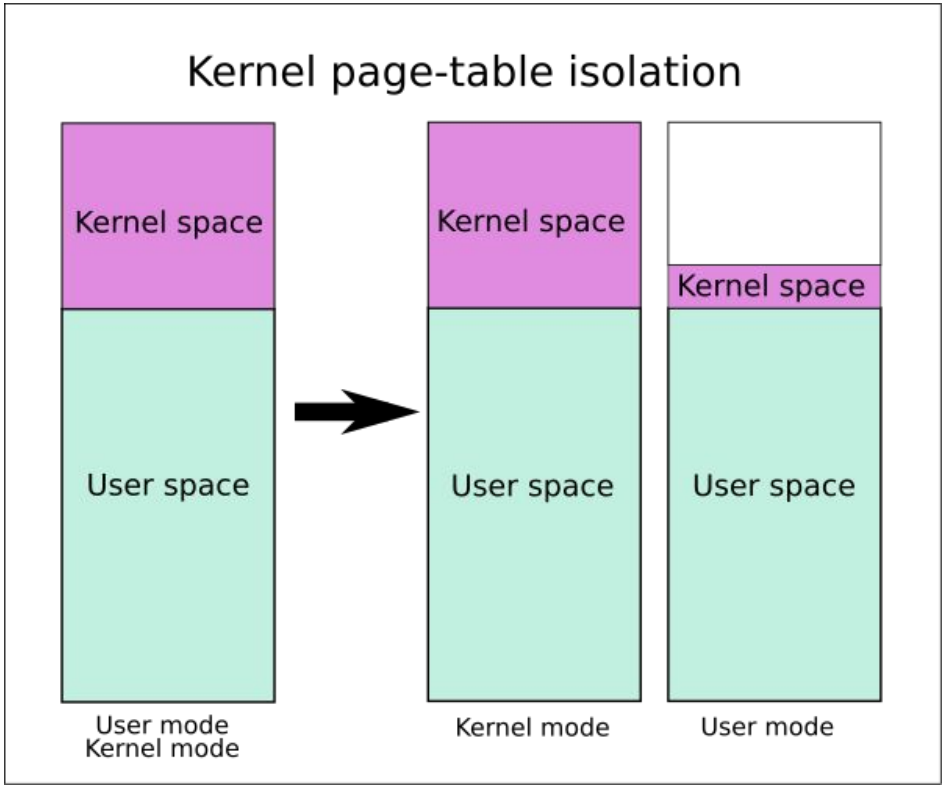
Meltdown Mitigation

- KPTI implements two page tables for each process. One is essentially unchanged and includes both kernel-space and user-space addresses, and is only used when the system is running in kernel mode.



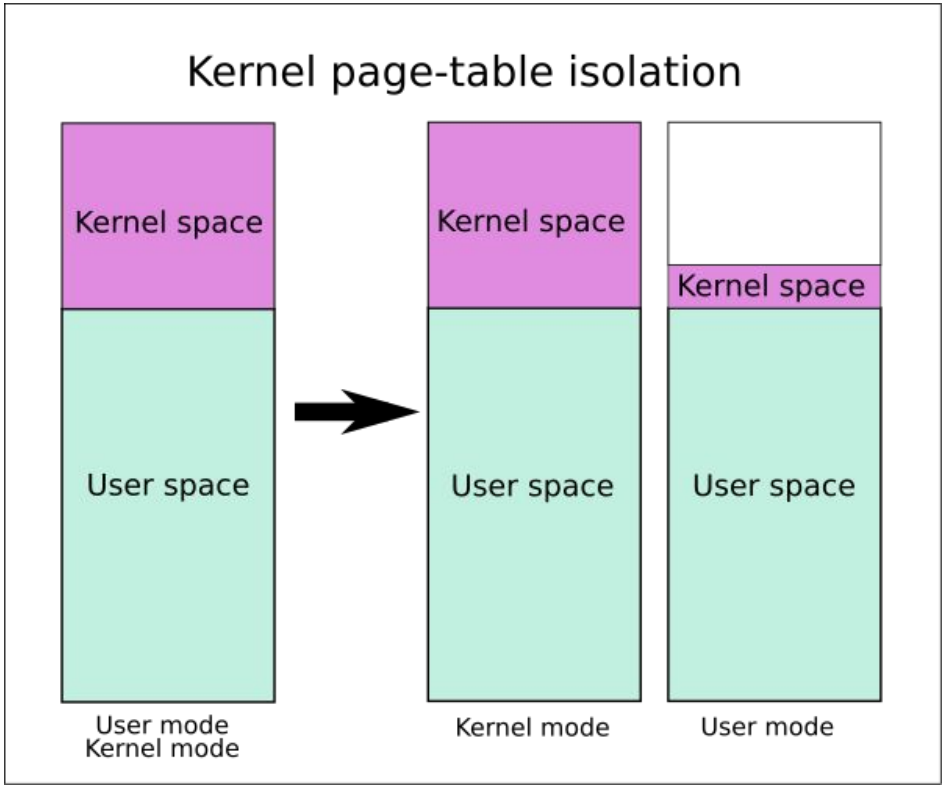
Meltdown Mitigation

- The second "shadow" page table contains a copy of all of the user-space mappings, but leaves out the kernel side. Instead, there is a minimal set of kernel-space mappings that provides the information needed to handle system calls and interrupts, but no more.



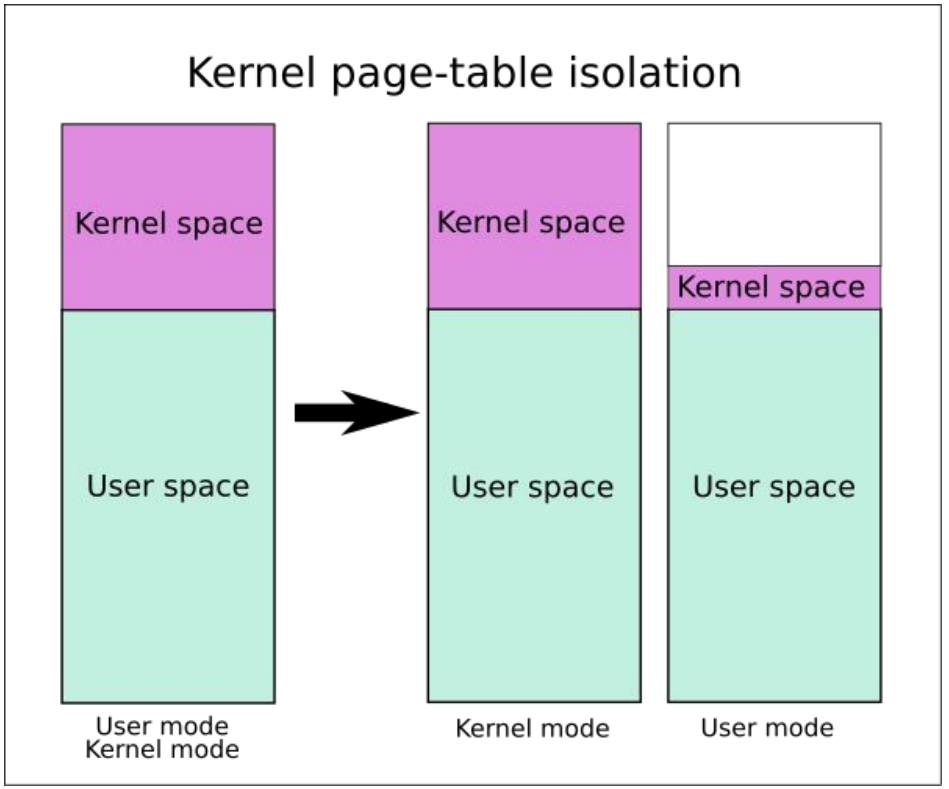
Meltdown Mitigation

- Whenever a process is running in user mode, the shadow page tables will be active. The bulk of the kernel's address space will thus be completely hidden from the process, defeating the known hardware-based attacks.



Meltdown Mitigation

- Whenever the system needs to switch to kernel mode, response to system call, exception, or interrupt, a switch to the other page tables will be made. The code that manages the return to user space must then make the shadow page tables active again.
- KASLR: kernel address space layout randomization
  Randomizes the location of the kernel address space on every boot



Spectre Mitigation

- Spectre is harder to exploit than Meltdown, but it is also harder to mitigate. However, it is possible to prevent specific known exploits based on Spectre through software patches.

- Remember to update your Operating System and Softwares to the newest version for keeping known Spectre attack away.

Spectre Mitigation

- Preventing Speculative Execution:
  - Ensure control flow leads the instruction
  - Software using serialization or speculation blocking
  - Causing a significant degradation in the performance
- Preventing Access to Secret Data (more for JIT compiler)
  - Chrome: each website per process
- Limiting Data Extraction from Covert Channels
- Preventing Branch Poisoning

Spectre Mitigation

- Preventing Data from Entering Covert Channels
  - Future processors (no such design is currently available)
- KAISER/KPTI does not help for Mitigation
- Google also have posted a patch called Retpoline for mitigating Spectre Variant 2
- Other Linux Spectre mitigation details: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html#turning-on-mitigation-for-spectre-variant-1-and-spectre-variant-2

Command line code for checking the vulnerabilities

To see if the computer(Linux) has the meltdown and spectre vulnerabilities:

$ git clone https://github.com/speed47/spectre-meltdown-checker.git

We can see there are still some Variants of Spectre are not solved.

Reference

https://meltdownattack.com/
https://meltdownattack.com/meltdown.pdf
https://spectreattack.com/spectre.pdf
https://searchdatacenter.techtarget.com/definition/out-of-order-execution
https://www.computerhope.com/jargon/s/spec-exec.htm
https://www.blackhat.com/docs/asia-17/materials/asia-17-Irazoqui-Cache-Side-Channel-Attack-Exploitability-And-Countermeasures.pdf
https://www.mikelangelo-project.eu/2016/09/cache-based-side-channel-attacks/
https://conference.hitb.org/hitbsecconf2016ams/materials/D2T1%20-%20Anders%20Fogh%20-%20Cache%20Side%20Channel%20Attacks.pdf
https://hackernoon.com/a-simplified-explanation-of-the-meltdown-cpu-vulnerability-ad316cd0f0de
http://www.cs.toronto.edu/~arnold/427/18s/427_18S/indepth/spectre_meltdown/index.html
http://www.cs.toronto.edu/~arnold/427/19s/427_19S/indepth/sm/Meltdown-and-Spectre.pdf
https://events19.linuxfoundation.org/wp-content/uploads/2017/11/Spectre-Meltdown-Linux-Greg-Kroah-Hartman-The-Linux-Foundation.pdf
https://lwn.net/Articles/738975/