Side-Channel Security

Chapter 4: Transient-Execution Attacks - Meltdown, Spectre & More

Daniel Gruss

March 20, 2025

Graz University of Technology

- Meltdown[4] and Spectre [2] are two CPU vulnerabilities
- Discovered in 2017 by 4 independent teams
- Due to an embargo, released at the beginning of 2018
- News coverage followed by a lot of panic


SECURITY FLAW REVEALED
Intel (Prev)
45.26
-1.59
[-3.39%]
Intel (After Hours)
44.85
-0.41
[-0.91%]
CAPITAL CONNECTION
SHROUT: ISSUE NOT UNIQUE TO INTEL, BUT IT'S AFFECTED THE MOST
CNBC
NETWORK

A lot of confusion fueled the panic

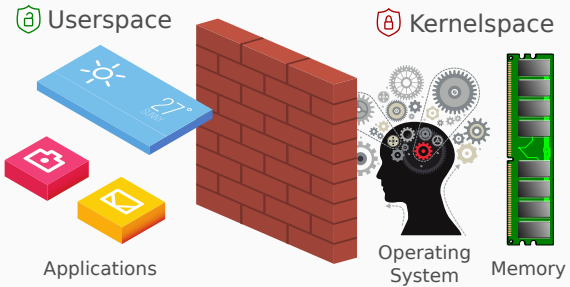
- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?
- Can the vulnerabilities be exploited remotely?
- What data is at risk?
- How hard is it to exploit the vulnerabilities?
- Is it already exploited?

Let's try to clarify these questions

MELTDOWN

Hardware Isolation

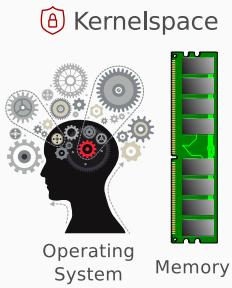
- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel
- There is only a well-defined interface → syscalls



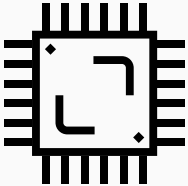
Meltdown Briefing

- Breaks isolation between applications and kernel
- User applications can access kernel addresses
- Entire physical memory is mapped in the kernel

→ Meltdown can read whole DRAM

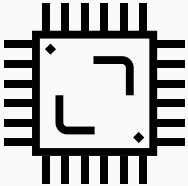


Meltdown Requirements

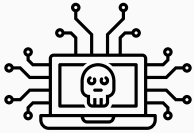


- Only on Intel CPUs and some ARMs (e.g. Cortex A15,A57,A72,A75)
- AMD and other ARMs seem to be unaffected
- Common cause: permission check done in parallel to load instruction
- Race condition between permission check and dependent operation(s)

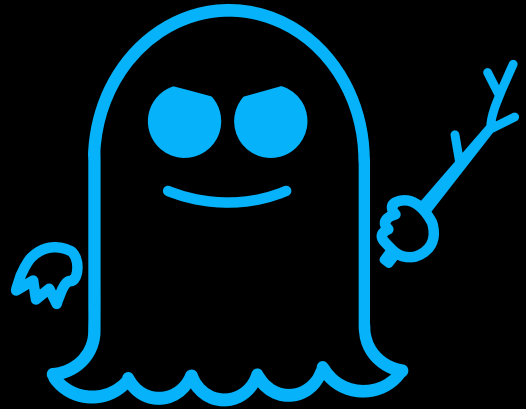
Meltdown Variant Requirements



- Meltdown variant: read privileged registers
- Limited to some registers, no memory content
- Reported by ARM
- Affects some ARMs (Cortex A15, A57, and A72)



- Meltdown requires code execution on the device (e.g. Apps)
- Untrusted code can read entire memory of device
- Cannot be triggered remotely
- Proof-of-concept code available online
- No info about environment required $\rightarrow$ easy to reproduce

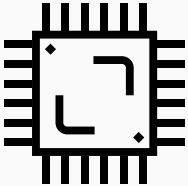


SPECTRE

- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
- Can also mistrain indirect calls

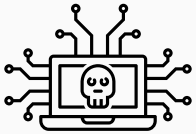
→ Spectre “convinces” program to execute code

Spectre Requirements



- On Intel and AMD CPUs
- Some ARMs (Cortex R and Cortex A) are also affected
- Common cause: speculative execution of branches
- Speculative execution leaves microarchitectural traces which leak secret

Spectre Exploitability



- Spectre (typically) requires code execution on the device (e.g. Apps)
- Untrusted code can convince trusted code to reveal secrets
- Can be triggered remotely (e.g. in the browser, NetSpectre)
- Proof-of-concept code available online
- Info about environment required $\rightarrow$ hard to reproduce

Background

Out-of-order Execution

7. Serve with cooked and peeled potatoes

Wait for an hour

LATENCY

Dependency

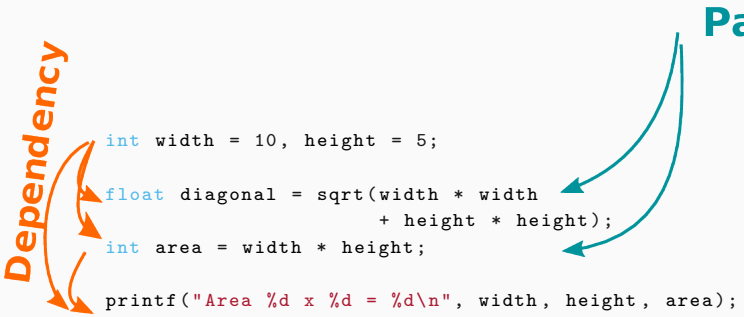
1. Wash and cut vegetables

2. Pick the basil leaves and set aside

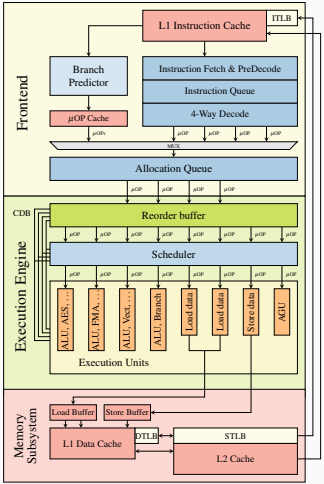
3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened

Parallelize

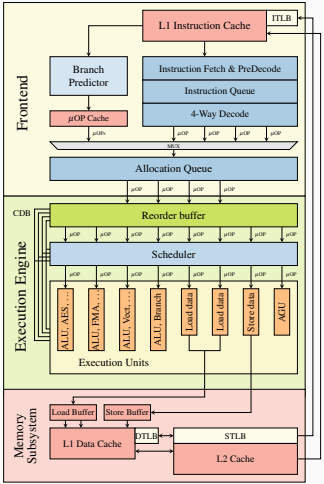

Parallelize
Dependency
int width = 10, height = 5;
float diagonal = sqrt(width * width
+ height * height);
int area = width * height;
printf("Area %d x %d = %d\n", width, height, area);

Out-of-order execution



- Instructions are fetched and decoded in the front-end
- Instructions are dispatched to the backend
- Instructions are processed by individual execution units

Out-of-order execution



- Instructions are executed out-of-order
- Instructions wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- Instructions retire in-order
  - State becomes architecturally visible

We are ready for the gory details of Meltdown

- Find something human readable, e.g., the Linux version

```
# sudo grep linux_banner /proc/kallsyms
ffffffff81a000e0 R linux_banner
```

Building the Code

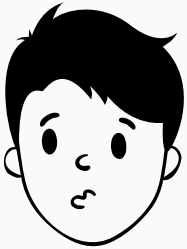
```
char data = *(char*) 0xffffffff81a000e0;
printf("%c\n", data);
```

- Compile and run

```
segfault at ffffffff81a000e0 ip
   0000000000400535
      sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are of course not accessible
- Any invalid access throws an exception → segmentation fault

- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue
- Then we can read the value
- Sounds like a good idea



- Still no kernel memory
- Maybe it is not that straight forward
- Privilege checks seem to work
- Are privilege checks also done when executing instructions out of order?
- Problem: out-of-order instructions are not visible

- Adapted code

```
*(volatile char*) 0;
array[0] = 0;
```

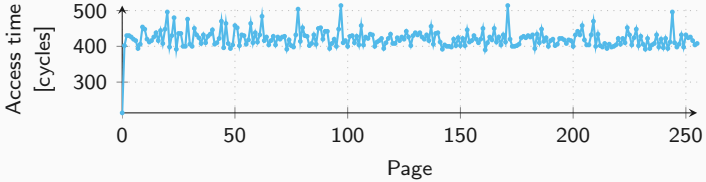
- `volatile` because compiler was not happy

```
warning: statement with no effect [-Wunused-value]
          *(char*) 0;
```

- Static code analyzer is still not happy

```
warning: Dereference of null pointer
         *(volatile char*)0;
```

- Flush+Reload over all pages of the array



- "Unreachable" code line was actually executed
- Exception was only thrown afterwards

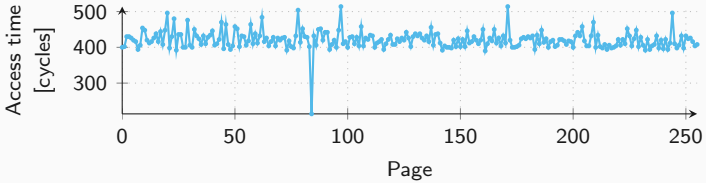
- Out-of-order instructions leave microarchitectural traces
- We can see them for example in the cache
- Give such instructions a name: transient instructions
- We can indirectly observe the execution of transient instructions

- Combine the two things

```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

- Then check whether any part of `array` is cached

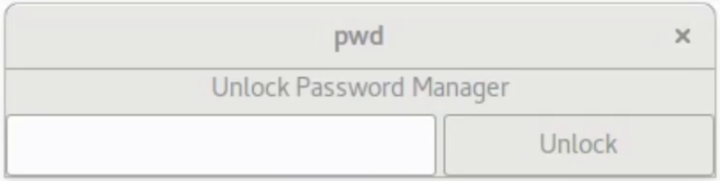
- Flush+Reload over all pages of the array



- Index of cache hit reveals data
- Permission check is in some cases not fast enough

- Using out-of-order execution, we can read data at (almost) any address*
- Privilege checks are sometimes too slow
- Allows to leak kernel memory
- Entire physical memory is typically also accessible in kernel address space


I SHIT YOU NOT
THERE WAS KERNEL MEMORY ALL OVER THE TERMINAL



Terminal

File Edit View Search Terminal Help

```
mschwarz@lab06:~/Documents$
```

```
e01d8150: 69 6c 69 63 6f 6e 20 47  72 61 70 68 69 63 73 2c |ilicon Graphics,|
e01d8160: 20 49 6e 63 2e 20 20 48  6f 77 65 76 65 72 2c 20 | Inc.  However, |
e01d8170: 74 68 65 20 61 75 74 68  6f 72 73 20 6d 61 6b 65 |the authors make|
e01d8180: 20 6e 6f 20 63 6c 61 69  6d 20 74 68 61 74 20 4d | no claim that M|
e01d8190: 65 73 61 0a 20 69 73 20  69 6e 20 61 6e 79 20 77 |esa. is in any w|
e01d81a0: 61 79 20 61 20 63 6f 6d  70 61 74 69 62 6c 65 20 |ay a compatible |
e01d81b0: 72 65 70 6c 61 63 65 6d  65 6e 74 20 66 6f 72 20 |replacement for |
e01d81c0: 4f 70 65 6e 47 4c 20 6f  72 20 61 73 73 6f 63 69 |OpenGL or associ|
e01d81d0: 61 74 65 64 20 77 69 74  68 0a 20 53 69 6c 69 63 |ated with. Silic|
e01d81e0: 6f 6e 20 47 72 61 70 68  69 63 73 2c 20 49 6e 63 |on Graphics, Inc|
e01d81f0: 2e 0a 20 2e 0a 20 54 68  69 73 20 76 65 72 73 69 |.. .. This versi|
e01d8200: 6f 6e 20 6f 66 20 4d 65  73 61 20 70 72 6f 76 69 |on of Mesa provi|
e01d8210: 64 65 73 20 47 4c 58 20  61 6e 64 20 44 52 49 20 |des GLX and DRI |
e01d8220: 63 61 70 61 62 69 6c 69  74 69 65 73 3a 20 69 74 |capabilities: it|
e01d8230: 20 69 73 20 63 61 70 61  62 6c 65 20 6f 66 0a 20 | is capable of. |
e01d8240: 62 6f 74 68 20 64 69 72  65 63 74 20 61 6e 64 20 |both direct and |
e01d8250: 69 6e 64 69 72 65 63 74  20 72 65 6e 64 65 72 69 |indirect renderi|
e01d8260: 6e 67 2e 20 20 46 6f 72  20 64 69 72 65 63 74 20 |ng.  For direct |
e01d8270: 72 65 6e 64 65 72 69 6e  67 2c 20 69 74 20 63 61 |rendering, it ca|
e01d8280: 6e 20 75 73 65 20 44 52  49 0a 20 6d 6f 64 75 6c |n use DRI. modul|
```

- Basic Meltdown code leads to a crash (segfault)
- How to prevent the crash?

Fault Handling

Fault Suppression

Fault Prevention

- Intel TSX to suppress exceptions instead of signal handler

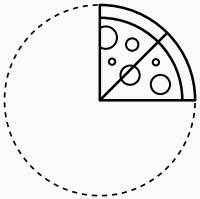
```
if(xbegin() == XBEGIN_STARTED) {
  char secret = *(char*) 0xffffffff81a000e0;
  array[secret * 4096] = 0;
  xend();
}

for (size_t i = 0; i < 256; i++) {
  if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
    printf("%c\n", i);
  }
}
```

- Speculative execution to prevent exceptions

```
int speculate = rand() % 2;
size_t address = (0xffffffff81a000e0 * speculate) +
                 ((size_t)&zero * (1 - speculate));
if(!speculate) {
  char secret = *(char*) address;
  array[secret * 4096] = 0;
}

for (size_t i = 0; i < 256; i++) {
  if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
    printf("%c\n", i);
  }
}
```


SO YOU ARE TELLING ME
YOU CAN DUMP THE MEMORY STORED IN L1?
imgflip.com


WHAT IF I TOLD YOU
YOU CAN LEAK THE ENTIRE MEMORY
imgflip.com

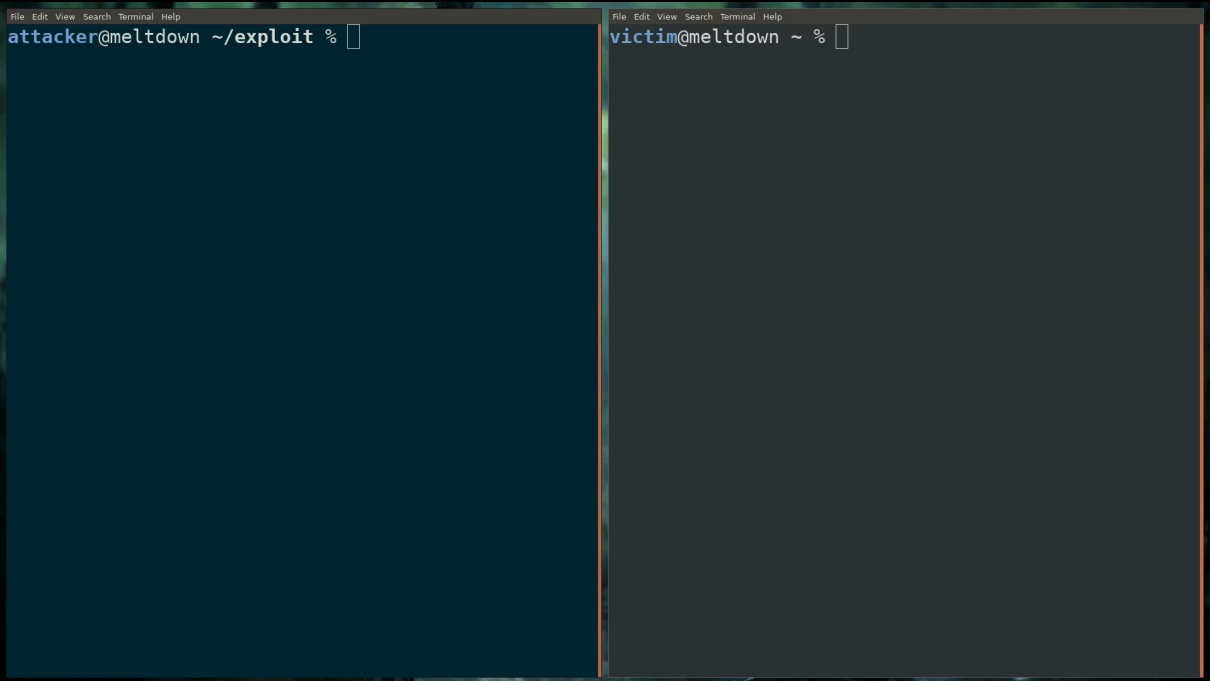


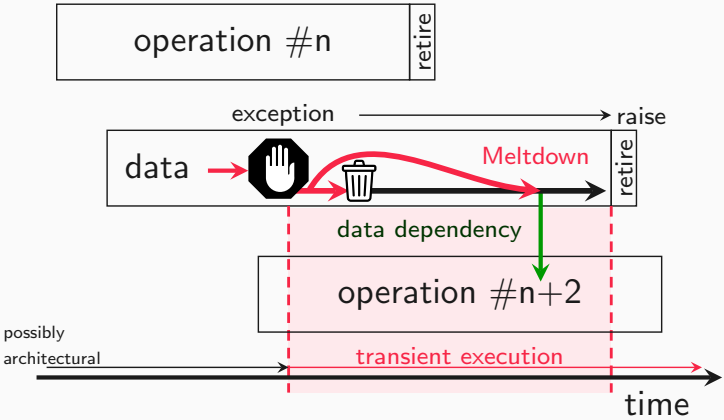
- Initial assumption: we can only read data stored in the L1 with Meltdown. Sort of:
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We still leak the data slowly, why?
- → *Original* Meltdown only leaks from the L1, but we can get data there with load gadgets [7]


I'LL JUST QUICKLY DUMP THE ENTIRE MEMORY VIA MELTDOWN
imgflip.com

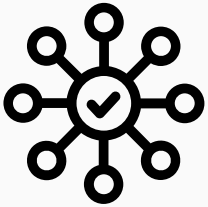
- Dumping the entire physical memory takes some time
  - Not very practical in most scenarios
- Can we mount more targeted attacks?

- Open-source utility for disk encryption
- Fork of TrueCrypt
- Cryptographic keys are stored in RAM
  - With Meltdown, we can extract the keys from DRAM

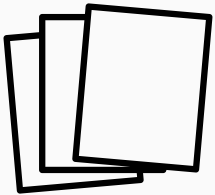

File Edit View Search Terminal Help
attacker@meltdown ~/exploit %
File Edit View Search Terminal Help
victim@meltdown ~ %


operation #n
retire
exception
raise
data
Meltdown
retire
data dependency
operation #n+2
possibly
architectural
transient execution
time

Generalization

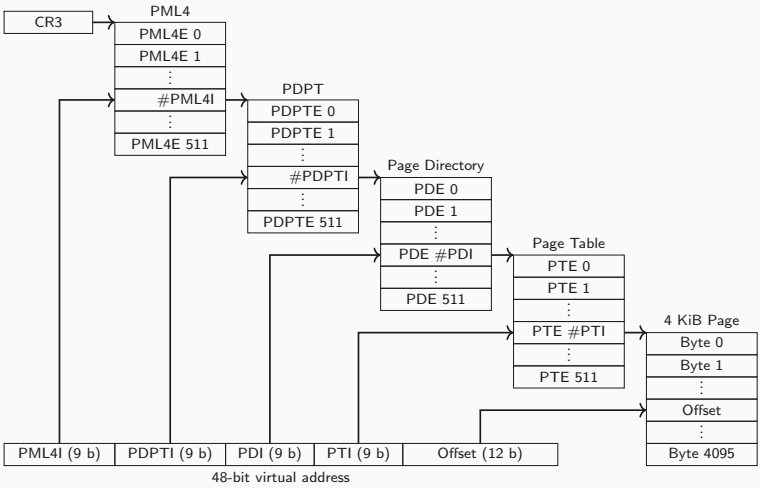


- Meltdown is a whole category of vulnerabilities
- Not only the user-accessible check
- Looking closer at the check...



- CPU uses virtual address spaces to isolate processes
- Physical memory is organized in page frames
- Virtual memory pages are mapped to page frames using page tables

Address Translation on x86-64

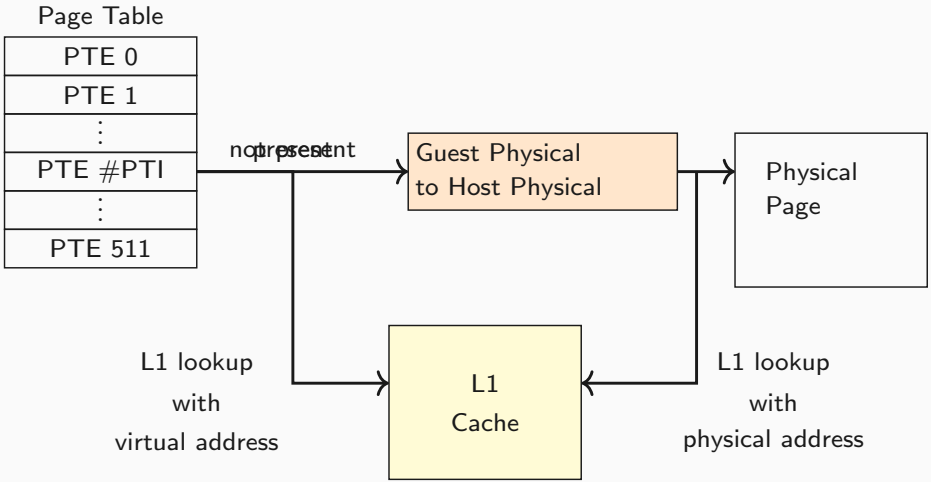


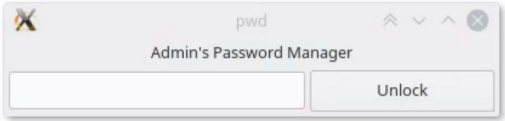
- User/Supervisor bit defines in which privilege level the page can be accessed

- Present bit is the next obvious bit

Foreshadow-NG [8]

- An even worse bug → Foreshadow-NG/L1TF
- Exploitable from VMs
- Allows leaking data from the L1 cache
- Same mechanism as Meltdown
- Just a different bit in the PTE


Page Table
PTE 0
PTE 1
PTE #PTI
PTE 511
not present
present
Guest Physical
to Host Physical
Physical
Page
L1 lookup
with
virtual address
L1
Cache
L1 lookup
with
physical address

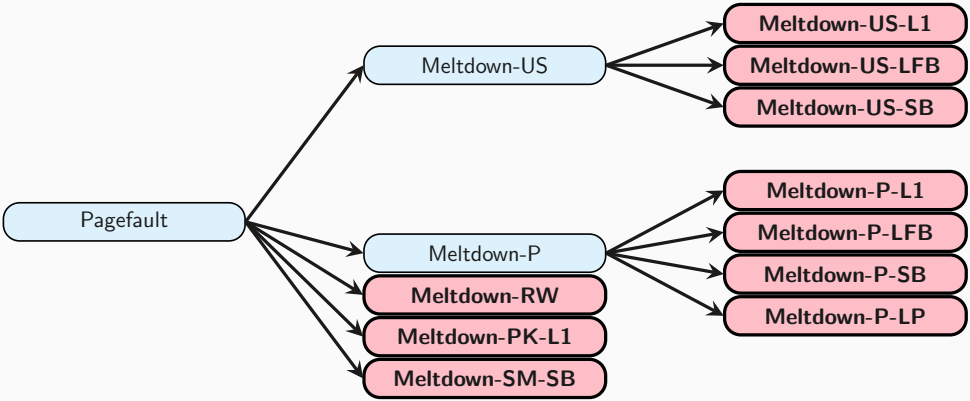


QEMU

```
Booting from ROM...
early console in extract_kernel
input_data: 0x0000000001e0a276
input_len: 0x000000000003d48f8
output: 0x0000000001000000
output_len: 0x00000000011bc258
kernel_total_size: 0x0000000000dee000
booted via startup_32()
Physical KASLR using RDTSC...
Virtual KASLR using RDTSC...

Decompressing Linux... Parsing ELF... Performing relocations... done.
Booting the kernel.
```

```
Run reader <pfn> [<cache miss threshold>] to leak hypervisor data from the L1

/ #
```

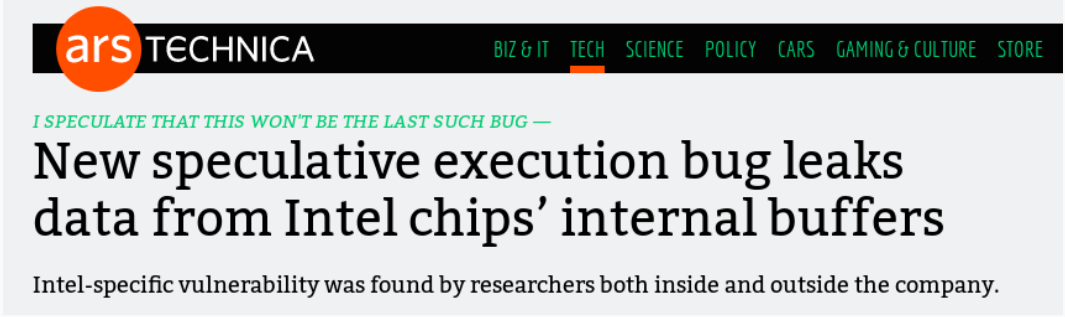
Meltdown Subtree: Exploiting Page-Table Bits



WIRED BUSINESS CULTURE GEAR IDEAS SCIENCE SECURITY TRANSPORTATION

Meltdown Redux: Intel Flaw Lets Hackers Siphon Secrets from Millions of PCs

Two different groups of researchers found another speculative execution attack that can steal all the data a CPU touches.



ars TECHNICA BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULTURE STORE

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.

- May 2019: 3 new Meltdown-type attacks
- Leakage from: line-fill buffer, store buffer, load ports
- **Key take-aways:**
  1. Leakage from various intermediate buffers ($\supset$ L1D)
  2. Transient execution through microcode assists ($\supset$ exceptions)

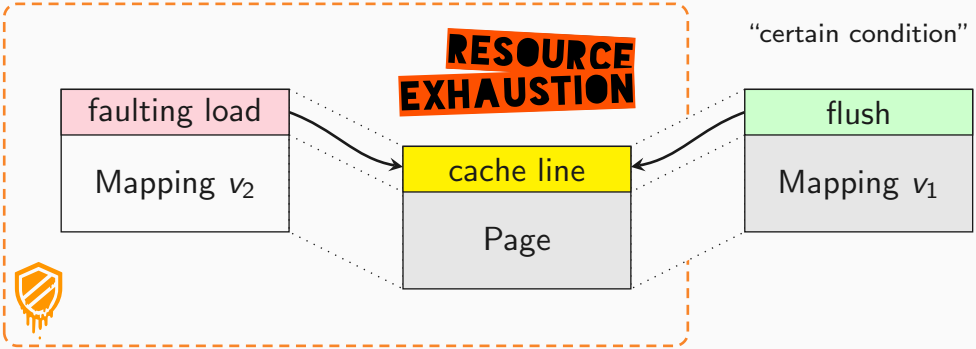
There is no noise. Noise is just someone else's data

Lemma 1: Noise is someone else's data

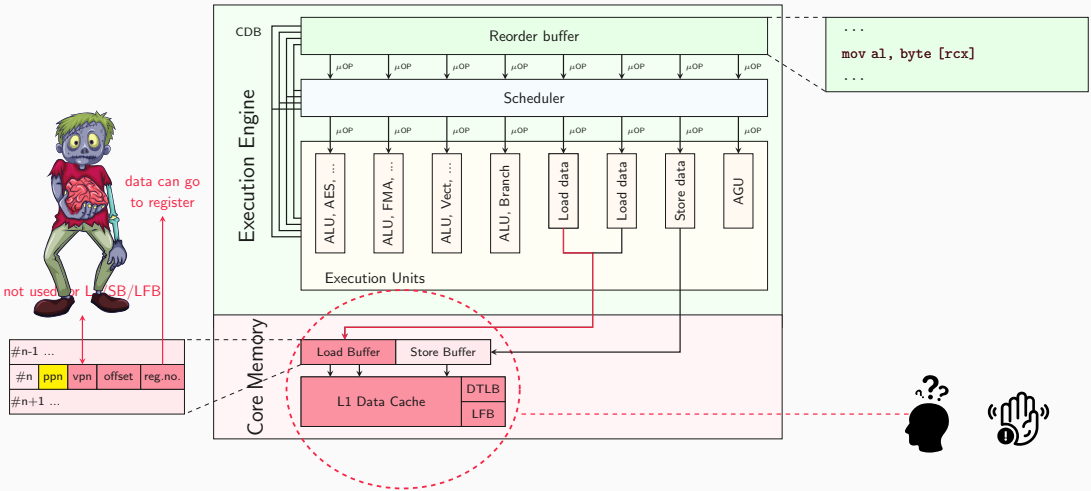
$$\lim_{\text{shield} \to \varnothing} \text{shield} = \text{fist}$$

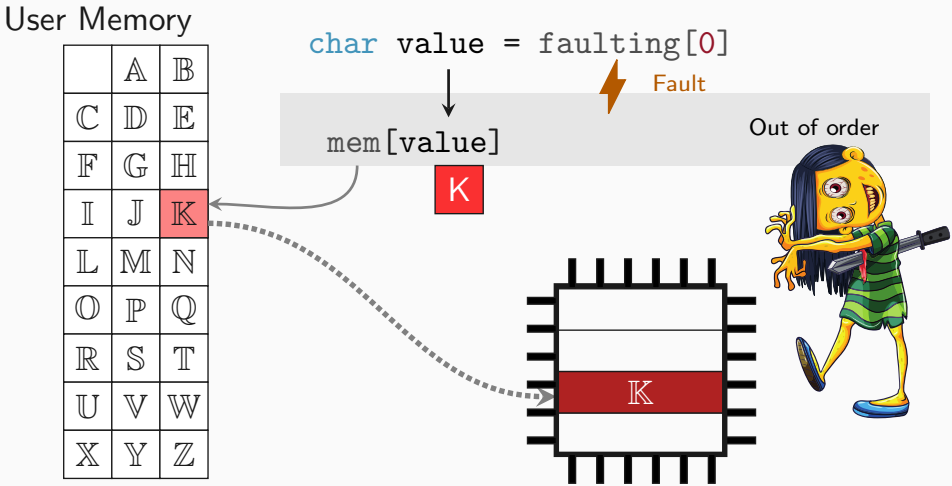
Deep Dive: Intel Analysis of Microarchitectural Data Sampling

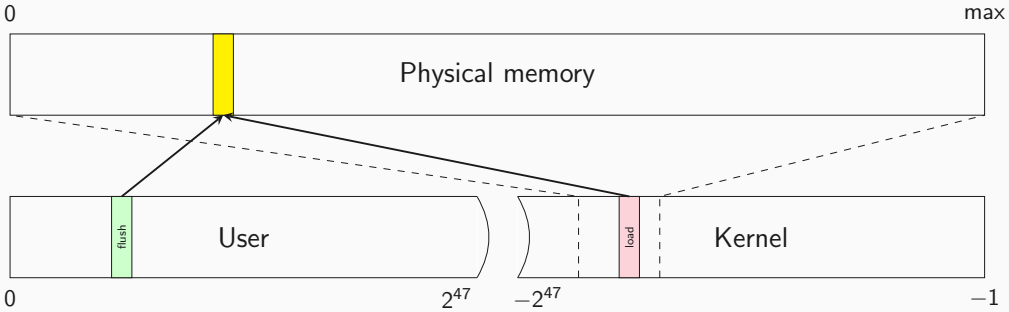
Fill buffers may retain stale data from prior memory requests until a new memory request overwrites the fill buffer. Under certain conditions, the fill buffer may speculatively forward data, including stale data, to a load operation that will cause a fault/assist.

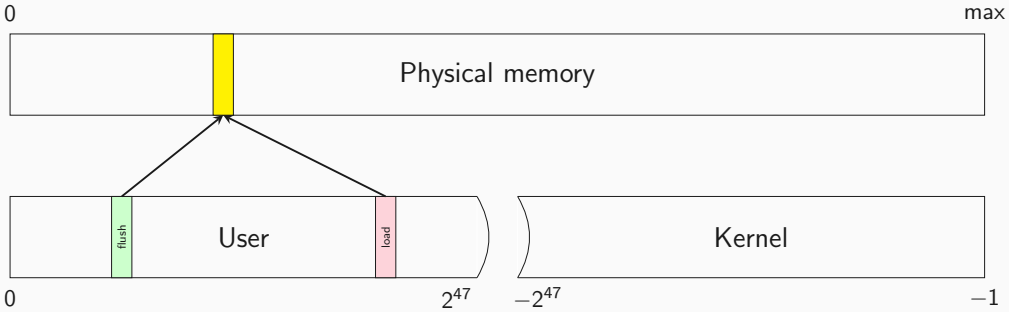

RESOURCE EXHAUSTION
"certain condition"
faulting load
Mapping v_2
cache line
Page
flush
Mapping v_1

Complex Load Situations

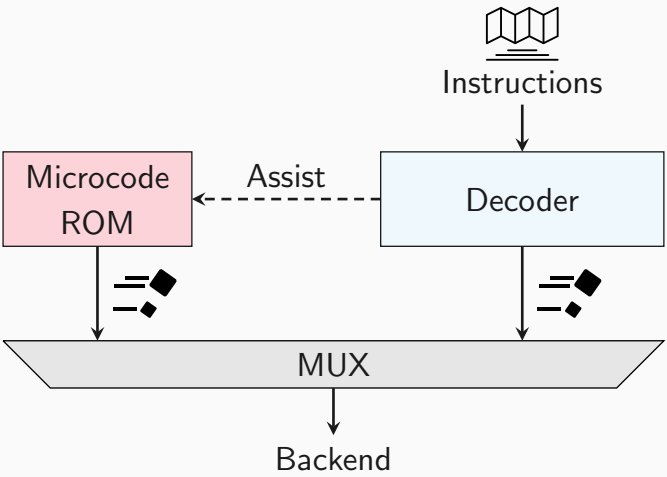


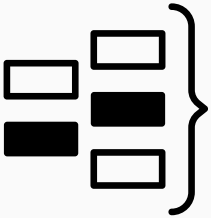

User Memory
char value = faulting[0]
Fault
mem[value]
Out of order
K
K


0
max
Physical memory
flush
User
load
Kernel
0
2^{47}
-2^{47}
-1


0
max
Physical memory
flush
User
load
Kernel
0
2^{47}
-2^{47}
-1

Microcode Assist (Variant 3)



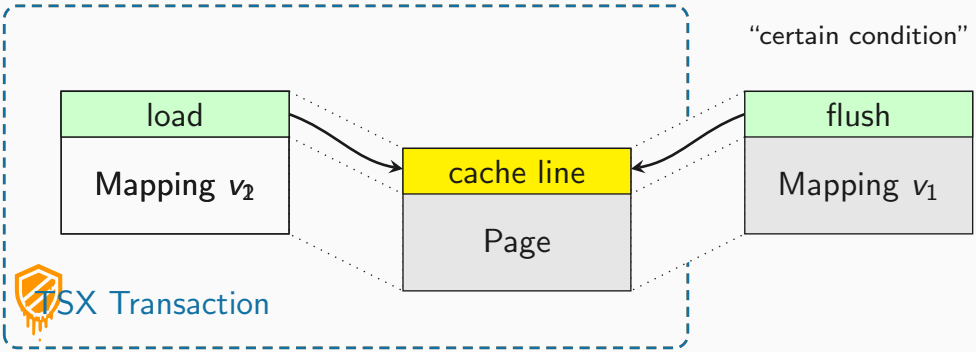


- Microcode assist handles rare cases

→ Microarchitectural fault

- Setting accessed/dirty bit in page table

→ Regularly reset on Windows


"certain condition"
load
Mapping v_2
cache line
Page
flush
Mapping v_1
TSX Transaction

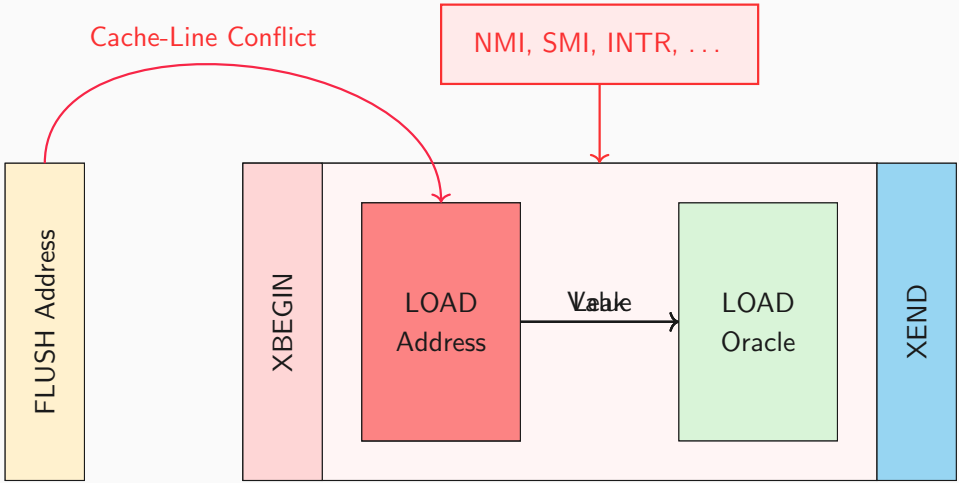
```
// Variant 2
flush(mapping);

if (xbegin() == _XBEGIN_STARTED) {
  maccess(lut + 4096 * mapping[0]);
  xend();
}
```

- **Data Conflicts**
- **Limited Transactional Resources**

- **Certain Instructions**
  - IO instructions, syscall, ...
- **Synchronous Exception Events**
  - #BR, #PF, #DB, #BP/INT3, ...

12.2.4.5 Miscellaneous Transactional Aborts

Asynchronous events (NMI, SMI, INTR, IPI, PMI, etc.) occurring during transactional execution may cause the transactional execution to abort and transition to a non-transactional execution. [...] For example, operating systems with timer ticks generate interrupts that can cause transactional aborts.


Cache-Line Conflict
NMI, SMI, INTR, . . .
FLUSH Address
XBEGIN
LOAD
Address
Value
LOAD
Oracle
XEND

Attack Targets

- Leak data on same and sibling hyperthread

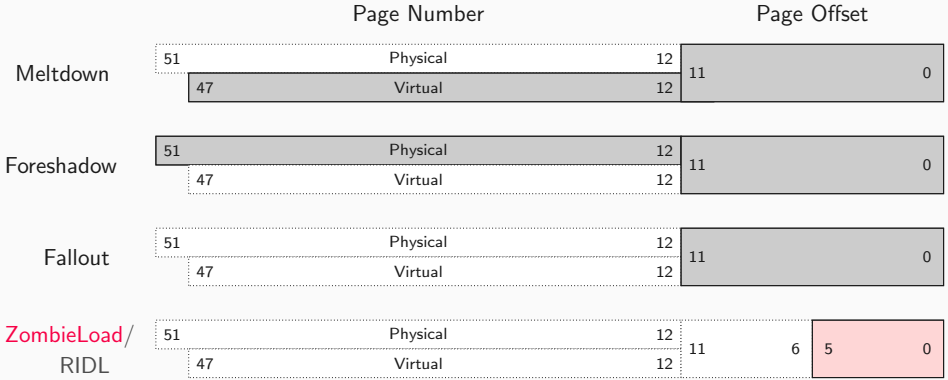
Applications

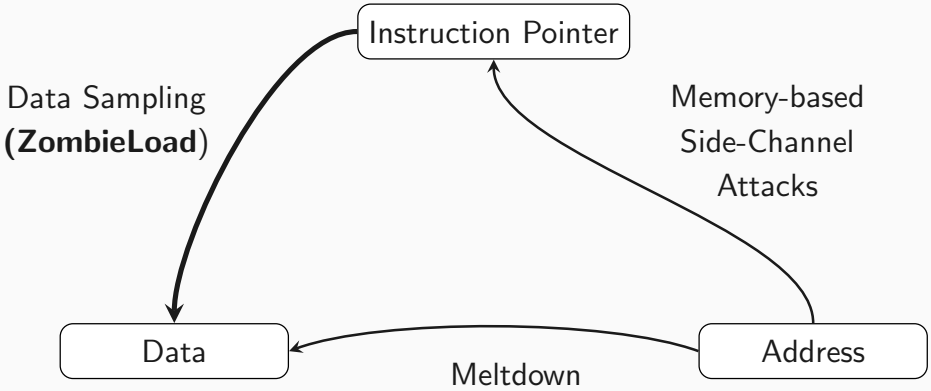
Operating System

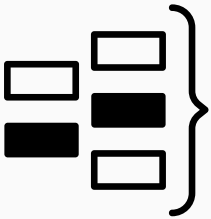
SGX Enclave

Virtual Machine

Hypervisor

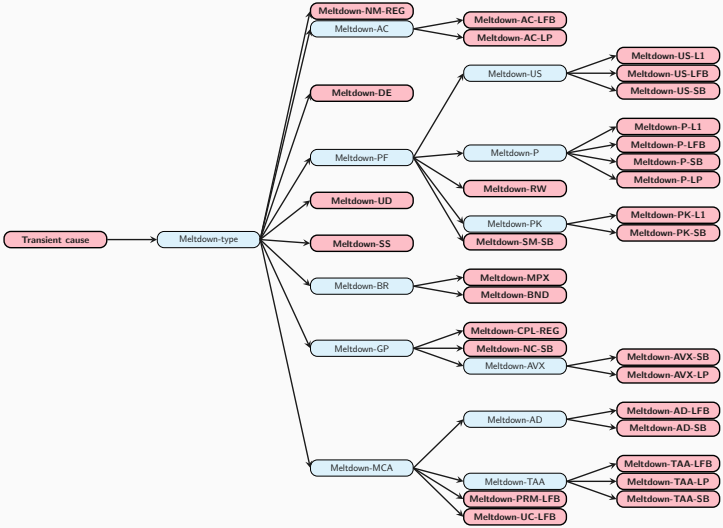

Page Number
Page Offset
Meltdown
51
Physical
12
47
Virtual
12
11
0
Foreshadow
51
Physical
12
47
Virtual
12
11
0
Fallout
51
Physical
12
47
Virtual
12
11
0
ZombieLoad/
RIDL
51
Physical
12
47
Virtual
12
11
6
5
0


Instruction Pointer
Data Sampling
(ZombieLoad)
Memory-based
Side-Channel
Attacks
Data
Address
Meltdown



- Optimization: only implement fast-path in silicon
- More complex edge cases (slow-path) in microcode
- Need help? Re-issue the load with a **microcode assist**
  - assist == "microarchitectural fault"
- Example: setting A/D bits in the page table walk
  - Likely many more!

Extended Meltdown tree with microcode assists



MELTDOWN

SPECTRE


PIZZA
SPECIAL RECIPES


»A table for 6 please«

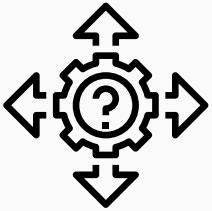
Speculative Cooking


»A table for 6 please«

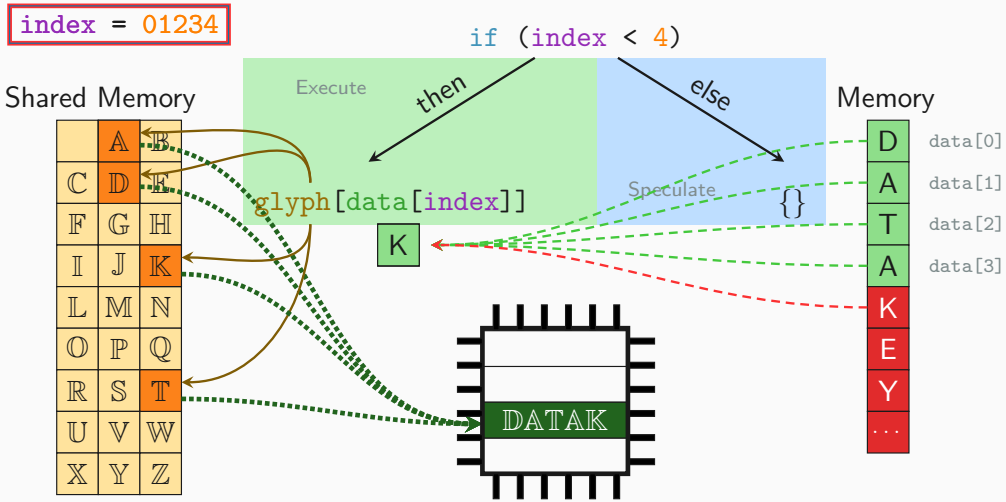

PIZZA
SPECIAL RECIPES


Pizza

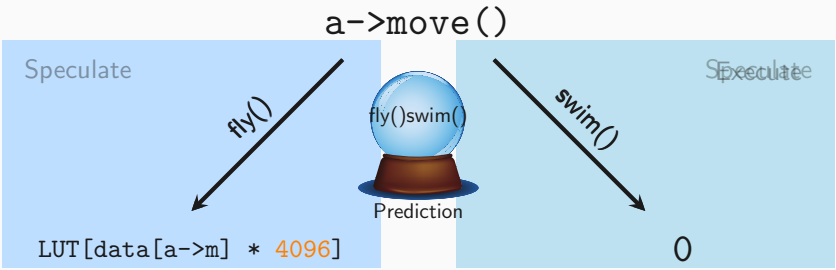
Speculation Causes



- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)
- Most are even shared among processes

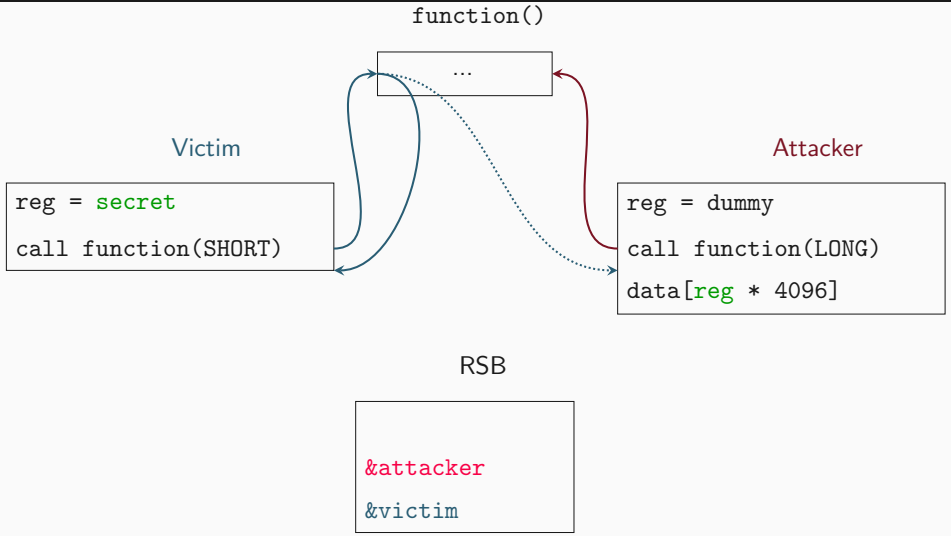

index = 01234
if (index < 4)
Shared Memory
Execute
then
else
Memory
glyph[data[index]]
Speculate
{}
K
data[0]
data[1]
data[2]
data[3]
DATAK

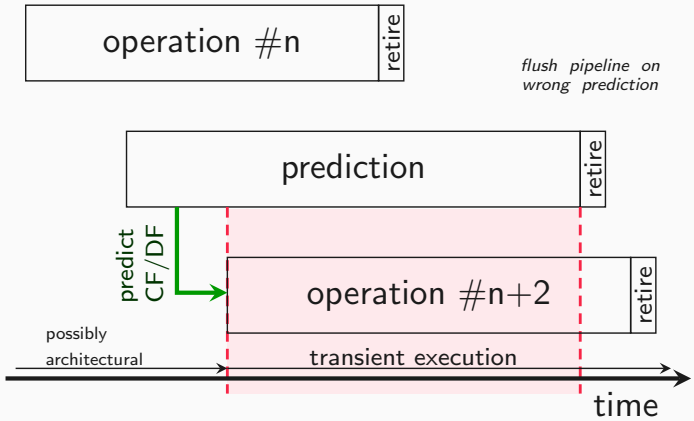
```
Animal* a = bird;
```



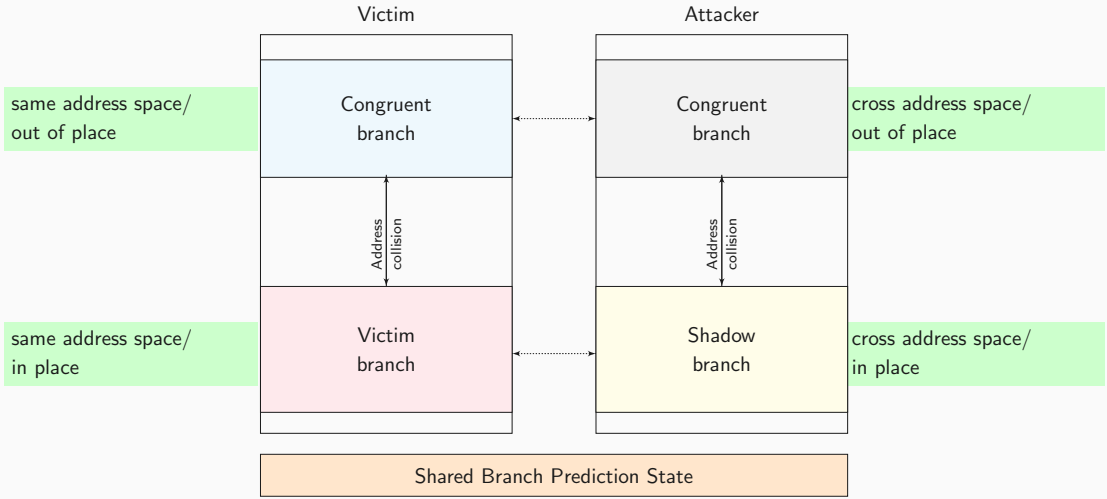
- Loads can be executed out-of-order → need to check for previous stores
- Check is time consuming
- Optimization: Speculate whether a store happened or not
  - no store: bypass check
  - stall

Spectre-RSB (Variant 5) [3, 5]




operation #n
retire
flush pipeline on wrong prediction
prediction
retire
predict CF/DF
operation #n+2
retire
possibly architectural
transient execution
time

Mistraining Location



- Well known: Race conditions & use-after-free
- Fix: Proper cleanup & locking
- What happens to locks in transient execution?

Thread 1:

```
func(A_ptr){
  mutex_lockmutex_lock(&Aptr->m);
  if(Aptr->Bptr->fptr){
    Aptr->Bptr->fptr(arg);
  }
  kfree(Aptr->Bptr);kfree(Aptr->Bptr);
  Aptr->Bptr = NULL;Aptr->Bptr = NULL;
  mutex_unlockmutex_unlock(&Aptr->m);
}
```

Thread 2:

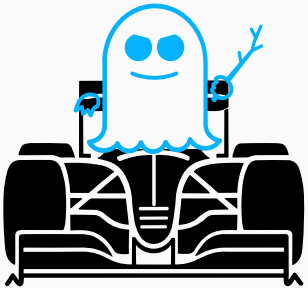
```
func(A_ptr){
  mutex_lockmutex_lock(&Aptr-
  if(Aptr->Bptr->fptr){
    Aptr->Bptr->fptr(arg);Aptr
  }
  kfree(Aptr->Bptr);
  Aptr->Bptr = NULL;
  mutex_unlockmutex_unlock(&Ap
}
```

- Challenge 1: How to bypass the mutex in thread 2?
- Challenge 2: How to change memory at Aptr→Bptr after `kfree` but before `NULL`?
  → IPI Storm: blast membarrier IPI to make
  victim core stop forever after `kfree` & before `NULL`

GhostRace: Challenge 1

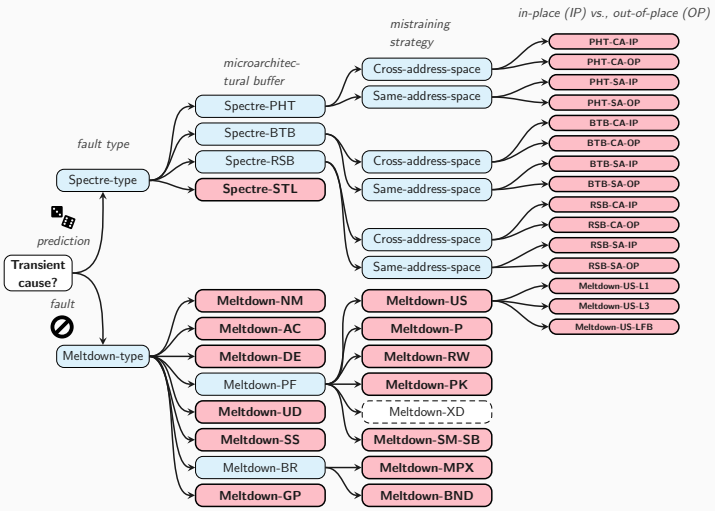
```
if (!__mutex_trylock_fast(lock))
  if(atomic_long_try_cmpxchg_acquire(&lock, ...))
    ↪arch_atomic_long_try_cmpxchg_acquire(&lock,,...)
      ↪arch_atomic_try_cmpxchg_acquire(&lock,,...)
        ↪arch_atomic_try_cmpxchg(&lock,,...)
          ↪arch_try_cmpxchg((&lock,, ...)
            ↪ __raw_try_cmpxchg(ptr, ...)({
              asm volatile( "lock cmpxchgq %2, %1"
              : "=a" (ret), "+m" (*ptr)
              : "r" (new), "0" (old)
              : "memory"
              );
          })
    return true;
```

- `lock cmpxchgq` is *atomic* not *serializing*
- We can speculate past it!



- Freeze thread between `kfree` and
- Fill memory with suitable code
- Make victim thread speculate past lock & execute chosen code
- Leak Data!

Recapping: Transient Execution Attacks



(The tree is even larger now, too large to show!)

	Leakage	Injection
Metadata		
Data		

	Leakage	Injection
Metadata		
Data		

We have ignored software side-channels for many many years:

- attacks on crypto $\rightarrow$ "software should be fixed"
- attacks on ASLR $\rightarrow$ "ASLR is broken anyway"
- attacks on SGX and TrustZone $\rightarrow$ "not part of the threat model"

$\rightarrow$ for years we solely optimized for performance

When you read the manuals...

After learning about a side channel you realize:

- the side channels were documented in the Intel manual
- only now we understand the implications

Conclusion

- Underestimated microarchitectural attacks for a long time
- Meltdown, Spectre and Foreshadow exploit performance optimizations
  - Allow to leak arbitrary memory
- CPUs are deterministic - there is no noise

Side-Channel Security

Chapter 4: Transient-Execution Attacks - Meltdown, Spectre & More

Daniel Gruss

March 20, 2025

Graz University of Technology

[1] Jann Horn. *speculative execution, variant 4: speculative store bypass*. 2018.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *S&P*. 2019.

[3] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. "Spectre Returns! Speculation Attacks using the Return Stack Buffer". In: *WOOT*. 2018.

[4] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security*. 2018.

[5] G. Maisuradze and C. Rossow. "ret2spec: Speculative Execution Using Return Stack Buffers". In: *CCS*. 2018.

[6] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-Privilege-Boundary Data Sampling". In: *CCS*. 2019.

[7] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. "Speculative Dereferencing of Registers: Reviving Foreshadow". In: *arXiv:2008.02307* (2020).

[8] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. *Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution*. 2018. URL: https://foreshadowattack.eu/foreshadow-NG.pdf.